MINIMUM SPANNING TREES
AND
ARBORESCENCES

By
RAMESH S. PATIL

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
AUGUST 1974

MINIMUM SPANNING TREES AND ARBORESCENCES

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
RAMESH S. PATIL

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
AUGUST 1974

CERTIFICATE

This is to certify that the thesis entitled, "MINIMUM SPANNING TREES AND ARBORESCENCES," is a record of the work carried out under my supervision and that it has not been submitted elsewhere for a degree.

Narsingh Deo

Kanpur
August 1974

Narsingh Deo
Professor of Electrical Engineering
and
Head, Computer Centre
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

CONTENTS

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. N. Deo for his invaluable guidance. He has been a source of constant inspiration throughout the period of this work.

I am also grateful to Messrs. M.S. Krishnamoorthy, V.G. Kane, A.B. Pai and A.S. Sethi for many fruitful discussions and help. The stimulating atmosphere of CC-202 is appreciated.

Thanks are also due to Mr. H.K. Nathani for diligently typing this thesis and the Computer Centre staff for their cooperation.

\- Ramesh S. Patil

Kanpur
July 1974

MINIMUM SPANNING TREES AND ARBORESCENCES

Abstract

Minimum Spanning Trees and Minimum Spanning Arborescence algorithms are required in many applications. A number of algorithms are available for finding minimum spanning trees. This thesis presents an analytical and empirical comparison of those algorithms. An algorithm for generating a minimum spanning arborescence, proof for its correctness, its complexity analysis, empirical study and implementation are also presented.

LIST OF SYMBOLS

G	Digraph
$V(G)$	Vertex set of G.
$E(G)$	Edge set of G.
n	Cardinality of vertex set $V(G)$
e	Cardinality of edge set $E(G)$.
v_i	A member of vertex set $V(G)$
$\langle v_i, v_j \rangle$	A member of edge set $E(G)$ of a digraph G.
(v_i, v_j)	A member of edge set $E(G)$ of an undirected graph G.
$w[\langle v_i, v_j \rangle]$	Weight of a directed edge $\langle v_i, v_j \rangle$.
$w[(v_i, v_j)]$	Weight of an undirected edge (v_i, v_j).
$w[G]$	Sum of the weights of the edges in a digraph (graph) G.
D	A subdigraph (subgraph)
T	A tree
c	A cycle
G_0	The 0^{th} reduced digraph which is same as the given digraph G.
G_i	The i^{th} reduced digraph of digraph G_0
G_i'	The minimum incidence subdigraph of G_i.
T_i	A spanning arborescence of G_i
S_i	A spanning arborescence of G_i containing $\lvert E(c) \rvert -1$ edges from every cycle c of G_i'.

H	The arborescence cover digraph of digraph G_o
M	The merger tree of digraph G_o
V_c	A composite vertex formed by merging the vertex set of cycle c.
G'_s	A connected component of G'_i.
T'_i	Intermediate spanning arborescence formed by merging the vertices in T_i forming the cycle c of G'_i alone.
S'_i	Intermediate spanning arborescence formed by merging the vertices in S_i forming the cycle c of G'_i alone.
We_i	Weight of the edge incident into the root of S_i in G'_i.

CHAPTER 1

INTRODUCTION

In the past four decades we have witnessed a steady development of graph theory, which in the last five to ten years has blossomed into a new period of intense activity. The main reason for this accelerated interest in graph theory is its demonstrated applications. Because of their intuitive diagrammatic representation, graphs have been found extremely useful in modeling systems arising in physical sciences, engineering and social sciences.

"Whenever graph theory is applied to any practical problem, it almost always leads to large graphs - graphs that are virtually impossible to analyse without the help of the computer. In fact, the high-speed digital computer is another reason for the recent phenomenal growth of interest in graph theory.

"Although the computers are very fast, they quickly reach their limit if used as a brute force to solve graph theory problems. Therefore, an algorithm must not only solve the problem correctly, but must do so efficiently. The two criteria for efficiency of an algorithm are the memory and the computation time requirements as a function of the size of the input." [9].

Perhaps the best known and most often used algorithms in graph theory are the spanning-tree algorithms. The spanning-tree algorithms

find use in various applications such as least-cost electrical wiring [42], minimum-cost communication and transportation network [41], minimum cost distribution network [41], network reliability problem [42], and the traveling salesman problem [26].

1.1 NOTATIONS AND DEFINITIONS

A directed graph or digraph is the ordered pair $G = \langle V(G), E(G) \rangle$ where $V(G)$ is a finite set $\{v_1, v_2, \ldots, v_n\}$ and $E(G)$ is a irreflexive relation in $V(G)$. The members of $V(G)$ are called vertices and the members of E, edges. The cardinality of $V(G)$ and $E(G)$ will be denoted respectively by n and e. Edge $\langle v_i, v_j \rangle$ is said to be incident out of vertex v_i and incident into vertex v_j. The vertices v_i and v_j are called the initial and terminal vertices of edge $\langle v_i, v_j \rangle$, respectively.

A directed graph G is said to be undirected graph or simply a graph, if $E(G)$ is symmetric. Edge (v_i, v_j) is said to be incident on vertices v_i and v_j. The vertices v_i and v_j are called the terminal vertices of edge (v_i, v_j).

In digraph G, if certain members of $E(G)$ can be placed in a sequence $\langle v_i, v_j \rangle, \langle v_j, v_k \rangle, \ldots, \langle v_s, v_t \rangle$, in which the first coordinate of each ordered pair is equal to the second coordinate of its predecessor in the sequence, then the set of edges in the sequence is a walk from v_i to v_t. The walk can also be written as a vertex sequence $(v_i, v_j, v_k, \ldots, v_s, v_t)$. If in this vertex sequence no vertex appears more than once, the walk is called a path. If the initial and the terminal vertices in a walk

are same and no other vertex appears more than once then the walk is called a <u>cycle</u>. Vertex v_i is said to be a <u>predecessor</u> of vertex v_t, if there exists a path from v_i to v_t and no path exists from v_t to v_i. Vertex v_t is called a <u>successor</u> of v_i.

A digraph D is said to be a subdigraph of a digraph G if $V(D)$ is a subset of $V(G)$ and $E(D)$ is a subset of $(V(D) \times V(D)) \cap E(G)$. D is said to be a <u>spanning</u> subdigraph of G if $V(D) = V(G)$ and is said to be an <u>induced</u> subdigraph if $E(D) = (V(D) \times V(D)) \cap E(G)$. Two subdigraphs D_1 D_2 of a digraph G are said to be <u>vertex disjoint</u> if $V(D_1) \cap V(D_2) = \emptyset$.

The <u>underlying undirected graph</u> of a digraph G is defined as an ordered pair $U = \langle V(U), E(V) \rangle$ where $V(U) = V(G)$ and $(v_i, v_j) \in E(U)$ if $\langle v_i, v_j \rangle$ or $\langle v_j, v_i \rangle \in E(G)$. An undirected graph is said to be <u>connected</u> if there exists a path between every pair of vertices. A directed graph is said to be <u>connected</u> if the underlying undirected graph is connected. A maximal, connected subgraph of a graph is called a <u>component</u> of the graph.

A <u>spanning tree</u> of graph G is a connected spanning subgraph without any cycles. A <u>spanning arborescence</u> of a digraph is a connected spanning subdigraph in which every vertex other than the root has exactly one edge incident into it.

A digraph G is said to be <u>weighted</u> digraph, if with every edge $\langle v_i, v_j \rangle$, there is associated a nonnegative real number $W[\langle v_i, v_j \rangle]$ $(W[(v_i, v_j)]$ for undirected graphs).

A spanning tree T of a weighted graph is said to be a minimum spanning tree (MST), if the sum of the weights of the edges in the tree is minimum over all the spanning trees of G.

A rooted spanning tree of a weighted graph is said to be minimum-path spanning tree (MPT), if the weight of the path from root to every other vertex is minimum over all such paths in the graph.

A spanning arborescene of a weighted digraph is said to be a minimum spanning arborescene (MSA) if the sum of the weights of the edges in the arborescence is minimum over all spanning arborescences of G.

A spanning arborescence of a weighted digraph is said to be minimum-path spanning arborescence (MPA) if the weight of the path from root to every other vertex is minimum over all directed paths in G.

1.2 SOME BASIC PROPERTIES OF SPANNING TREE

Some basic properties of spanning trees used in the present thesis are given below. For the proof of these properties we refer to Deo [9] and Harary [25].

Theorem 1.1. A graph G is called a spanning tree, if

(i) G is connected and cycle-less, or

(ii) G is connected and has n-1 edges, or

(iii) G is cycleless and has n-1 edges, or

(iv) There is exactly one path between every pair of vertices in G, or

(v) G is minimally connected.

Theorem 1.2. Every connected graph has atleast one spanning tree.

Theorem 1.3. In an arborescence, there is exactly one directed path from root to every other vertex.

Theorem 1.4. Every strongly connected digraph has atleast one spanning arborescence.

1.3 PLAN OF THE THESIS

Various algorithms for generation of minimum spanning trees, minimum-path spanning trees, minimum spanning arborescences and minimum-path spanning arborescence are described in Chapter 2, along with the modifications suggested for them, and their computational aspects are considered. The empirical result on the computation time and relative merits and demerits of the algorithms are also discussed.

On surveying the available MST algorithms it was observed that no existing MST algorithm could be suitably modified to yield minimum spanning arborescence. An algorithm for the MSA is developed in Chapter 3. The proof of correctness and computational aspects of the proposed algorithm are also included.

Some applications of spanning tree algorithms are briefly described in Chapter 4, and some areas of future work are suggested. FORTRAN implementation for a few selected algorithms are given in the Appendix.

CHAPTER 2

SPANNING TREE ALGORITHMS - A SURVEY

The first major contribution to minimum spanning tree (MST) problem was by J.B. Kruskal [34]. Kruskal, however, did not suggest any scheme for implementing his algorithm on a computer. Different implementations for the Kruskal algorithm have been suggested by Obruca [39], Seppanen [46] and McIlroy [37]. Another MST algorithm was suggested independently by R.C. Prim [42] and E.W. Dijkstra [12]. A recent analysis of MST algorithms by Kershunbaum and Van Slyke [30] and of the set merging algorithms by Hopcroft and Ullman [28] show that the Kruskal algorithm is more efficient than the Prim and Dijkstra algorithm when the input graph is sparse.. A modification to Kruskal's algorithm using heap sort [33] for picking the minimum weight edge and a similar modification to the Prim and Dijkstra algorithm using tree sort [33] suggested by Kershanbaum and Van Slyke [30] improve the performance of the algorithms considerably.

2.1 THE KRUSKAL ALGORITHM

Kruskal showed that an MST could be obtained by repeated applications of the following steps.

Take the smallest edge which has not been chosen or discarded. If it does not form a cycle with some of the previously chosen edges

add it to the chosen edges; otherwise, discard it.

The Kruskal algorithm is conceptually simple, but the means of implementing it on a computer is not obvious. The computer implementation was first suggested by A. Obruca [40]. In Obruca's implementation the graph is represented by a nxn matrix, called the weight matrix , where the $(i,j)^{th}$ element of the matrix represents the weight of the edge joining the vertices v_i and v_j.

During an iteration of the Kruskal algorithm the subgraph formed by the selected edges is a collection of subtrees. In Obruca's implementation the selection or rejection of an edge is done using a labeling procedure suggested by Loberman and Weinberger [37]. The labeling procedure is as follows:

Initialization: Initialize the predecessor of each vertex = 0. This makes each vertex a subtree. For the selection or rejection of picked edge (v_i,v_j) do the following:

Step 1: If the root of subtree containing $v_i \neq$ the root of subtree containing v_j, then go to Step 2, else go to Step 4.

Step 2: Change the direction of the edges in the path from v_i to the root of its subtree. This makes v_i the root of the subtree containing it.

Step 3: Make v_j the predecessor of v_i. (Graft the two subtrees.)

Step 4: Make $w(v_i,v_j) = \infty$.

Searching for the smallest entry in the lower triangle of the matrix requires computation of $O(n^2)$. The selection or rejection of an edge requires computation of $O(n)$. In the worst case the above steps are repeated for each element of the lower triangle of the matrix. Therefore, the execution time of Obruca's implementation is bounded by $O(n^4)$.

We now consider an implementation suggested by Seppanen [46]. In Seppanen's implementation the graph is represented by a list of edges (v_i, v_j), sorted in nondecreasing order of weights. During an iteration each subtree (i.e., connected component in the subgraph formed by selected edges), is identified by a component label. The labeling procedure for the four possible conditions arising in selection or rejection of the picked edge is described next.

Case 1. If neither of the two terminal vertices are included in a tree, the edge is taken as a new tree and vertices are labeled by an increased component number C.

Case 2. If one terminal vertex is in a tree and the other is not in any tree, the edge is added to the tree.

Case 3. If the two terminal vertices are on different trees, then these two trees are grafted into a single tree by relabeling the labels of the second tree.

Case 4. If both the terminal vertices are in the same tree, then this edge forms a cycle and is discarded.

A structured language [7] implementation of the algorithm is given next.

```
Procedure      SPTREE(v,c,F,H,p,T)

value:  v,c; Integer v,e,p; Integer array F,H,T.

begin: spanning tree
 c ← 0
 n ← 0
 for k = 1 step 1 until v do v(K) = 0
 for k = 1 step 1 until e do
  begin: loop 1
   i ← F(k)
   j ← H(k)
    if V(i) = 0 then do
     begin:
      T(k-n) ← k
       if V(j) = 0 then do
        begin: case I;
         c ← c+1
         V(i) ← o
         V(j) ← o
        end; case 1;
        else:: Case 2; V(j)←V(j);
       end;
      if V(i) ≠ V(j) then do
       begin: Case 3;
        T(k-n) ← k
        i ← V(i)
        j ← V(j)
        for r = 1 step 1 until v do
         begin:
          if V(r) = j then V(r)← i
         end;
       end; Case 3;
      else; n ← n+1
     end; loop 1;
   p ← v-c+n
  end; spanning tree;
```

The loop 1 of the program is executed once for every edge picked. For each iteration of the outer loop one of the four cases in labeling procedure is carried out. Case 1,2 and 4 require a constant computation. Case 3 requires computation linear in n, and in the worst case this may be executed n/2 times. Therefore, the upper bound on the computation is $O(\max(e \log_2 e, n^2))$. The sorting of edges require a computation time proportional to e log e. In actual computation it is observed that the e log e term dominates the computation, as the number of times case 3 is executed is much less than n/2.

Kershanbaum and Van Slyke [30] observed that in an average case the number of edges picked in the Kruskal algorithm is much smaller than e. Based on this observation they suggested a modification to Seppanen's implementation using heap sort [33]. This reduces the computation time required for sorting the edges to O(e + m log e) where m is the number of edges picked.

The problem of generating a spanning tree can also be formulated as a set merging problem [28]. An algorithm for grafting two subtrees into a tree using set merging was suggested by Knuth [32].

The analysis of the algorithm was given by Fischer [15], who showed that O(e log e) is a lower bound, and Peterson [39], who showed it to be the upper bound also. An improvement of the algorithm was suggested by McIlroy [37]. An implementation of McIlroy's algorithm is given below.

```
Procedure SPTREE(F,H,v,e,Edge,c,w)
Value: v,c; Integers: v,e,c,v_i,v_j,label_i,label_j
Integer Array: F(c), H(c), Edge(e),W(e), Pred(v), Num(v)
Global variables: Pred, Num, v,e.
begin: SPTREE
 begin: Initialization
 for l=1 step 1 until v do
  begin:
  Pred(l) ← 0
  Num(l) ← 1
  end;
 for l=1 step 1 until e do
  begin:
   Edge(l) ← 0
  end;
 iflg ← 0
 c ← n
 ne ← 0
end: Initialization
 begin:set merging
 If n e  < e and c > 1 then do
  begin:
   ne ← ne+1
   C: Procedure SORT returns the subscript of the smallest weight
      edge not yet picked.
   Call SORT (W,E,k)
   v_i ← F(k)
   v_j ← H(k)
   labeli ← Find (v_i)
   labelj ← Find (v_j)
   If label i ≠ label j then do
    begin:
     Call MERGE (labeli, labelj)
    end;
  end; set merging
 end; SPTREE
```

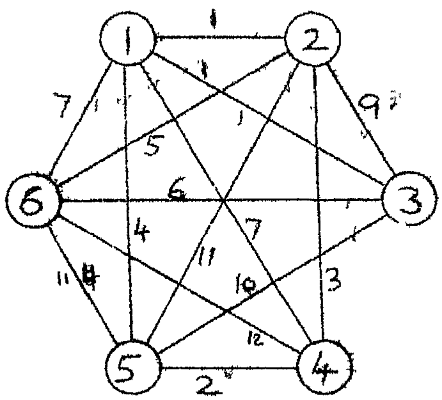
```
Procedure FIND(v_i)
Integer: top, stackp, find, label;
Integer Array: Pred(v), Num(v), stack(v);
Global variables: Pred, Num, e,v;
begin: find
 label <- v_i
 stackp <- 0
  while pred(label) > 0 do
   begin:
    stackp <- stackp + 1
    stack(stackp) <- label
    label <- Pred(label)
   end;
  while stackp > 1 do
   begin
    stackp <- stackp - 1
    top <- stack(stackp)
    Pred(top) <- label
   end;
end; find

Procedure MERGE(v_i,v_j)
Integers: idmy, v_i,v_j,v;
Integer Arrays: Pred(v), Num(v);
Global variables: Pred, Num, v,e;
begin; merge
 idmy <- Num(v_i) + Num(v_j)
 If Num(v_i) > Num(v_j) then
  begin;
   Num(v_i) <- idmy
   Pred(v_j) <- v_i
  end;
 else:
    begin:
     Num(v_j) <- idmy
      Pred(v_i) <- v_j
    end;
end; merge.
```

An example for the above algorithm is given below.

Let the F-H array for the input graph (Figure 2.1) be given as follows:

F:	1	1	4	2	1	2	3	1	1	2	3	2	5	6
H:	2	3	5	4	5	6	6	6	4	3	5	5	6	4

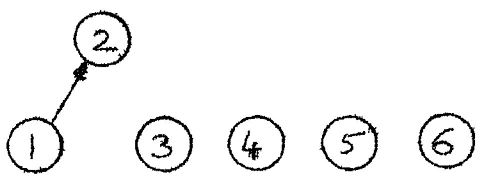


n = 6 e = 14

Figure 2.1

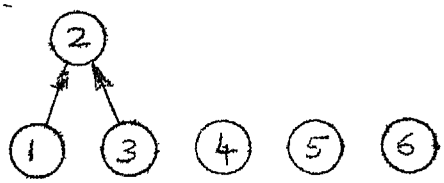
Then following are the steps in the course of algorithm:

Step 1: edge picked (1,2); selected.



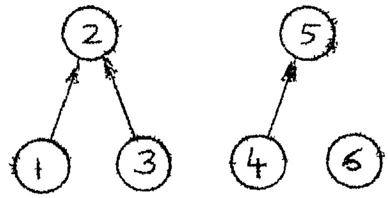
(a)

Step 2: Edge picked (1,3); selected



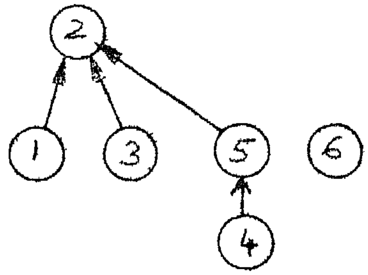
(b)

Step 3: Edge picked (4,5); selected



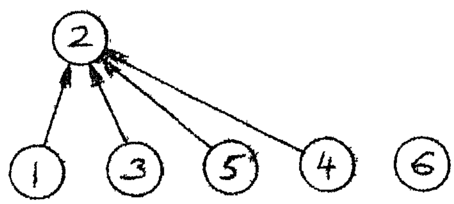
(c)

Step 4: Edge picked (2,4); selected.



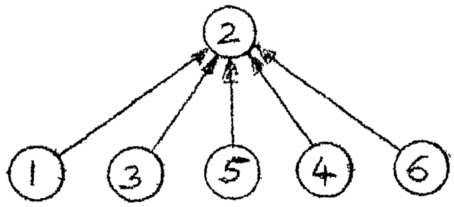
(d)

Step 5: Edge picked (1,5); rejected



(e)

Step 6: Edge picked (2,6); selected



(f)

Figure 2.2

The algorithm terminates at Step 6.

The MST formed is

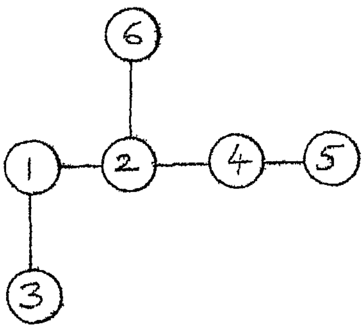


Figure 2.3

An analysis of the above algorithm was done by Hopcroft and Ullman [27]. They showed that the algorithm is bounded $O(e\ Z(n))$ where $Z(n)$ is defined for $n > 0$ to be the least number K for which $F(k) \geq n$, where,

$$F(0) = 1$$

$$\text{and} \quad F(i) = F(i-1)2^{F(i-1)} \quad \text{for } i \geq 1.$$

The first five values of F are 1,2,8,2024 and 2^{2059}. Z(n) is a slowly growing function and can be approximated to a constant. For graphs with 8 to 10^{687} vertices Z has a value between 3 and 5.

To obtain an MST the edges are picked in increasing order of weight and they are selected or rejected using the McIlory's algorithm. Sorting of edges using heap sort requires computation of O(e+n log e) where n is the number of edges picked. Therefore the Kruskel MST algorithm is bounded by O(e log e).

2.2 THE PRIM AND DIJKSTRA ALGORITHM

The Prim and Dijkstra MST algorithm [11,42] is based on the following theorem:

Theorem 2.1. A spanning tree T of graph G is an MST if and only if for every subtree $S \subset T$, there is in T an edge of smallest weight among all those connecting a vertex in S to a vertex in G-S [36].

In the algorithm the edges of the graph are divided into three sets.

Set 1: The edges assigned to the tree under construction.

Set 2: The edges from which the next edge is to be added to Set 1 will be selected (edges connecting the subtree formed by the edges in set I, to neighbouring vertices).

Set 3: The remaining edges (rejected or not yet rejected).

The vertices are divided in two sets.

Set A: The vertices connected by the edges in Set 1.

Set B: The remaining vertices.

The algorithm starts with any arbitrary vertex as set A, the starting vertex, and all edges incident on it as set 2. The MST is developed by adding one vertex to set A in each iteration of the following algorithm.

Step 1: The shortest edge in Set 2 is removed from the Set 2 and added to Set 1. As a result one vertex is transferred from Set B to Set A.

Step 2: Consider the edges leading from the vertex that has just been transferred to Set A, to the vertices that are still in Set B. If the edge under consideration is larger than the corresponding edge in Set 2, it is rejected, if it is shorter, it replaces the corresponding edge in Set 2, and later is rejected.

Step 3: If Set 2 and Set B are empty, stop, otherwise go to Step 1.

In the implementation, the vertices are given temporary and permanent labels. The vertices in Set A are given permanent labels. The vertices in Set B are given temporary labels denoting the weights of the edges (in Set 2) connecting them to vertices in Set A. The input graph is represented by an nxn weight matrix, where the (i,j)th element of the matrix represents the weight of the edge joining the vertices v_i and v_j. An implementation of this is given next.

Procedure MINTREE (D,n,s,Pred).

```
Value: n,s; Integers: p,n,s,z

Integer Array: D(n,n) Veet(n); Pred(n), Label(n)

begin:
 for l = 1 step 1 until n do
  begin:
   label(l) ← ∞
   Vect(l) ← 0
  end;
 label(s) ← 0
 Vect(s) ← 0
 i ← s
 P ← s
 while P ≠ 0 do
  begin:
   m ← ∞
   vect(p) ← 1
   p ← 0
  for j = 1 step 1 until n do
   begin:
    if vect (j) ≠ then do
C: The next statement is altered for shortest spanning tree.
     z ← D(i,j)
     if z ← label(j) then do
      begin:
       label(j) ← z
       Pred(j) ← i
       if label(j) ⩽ m then do
        begin:
         m ← label(j)
         p ← j
        end;
      end;
     end;
    end;
    i ← P
  end; spanning tree.
```

Each iteration requires computation proportional to n, and there are exactly n iterations. Therefore, the computation is bounded by $O(n^2)$.

A modification to the Prim and Dijkstra algorithm using tree sort was suggested by Kershenbaum and Van Slyke [30]. In the modified algorithm edges in set B are sorted during the first iteration using tree sort [33]. In every iteration, thereafter, some edges are added to the sorted tree (of binary tree sort) and the minimum weight edge is deleted from it. A FORTRAN implementation using the binary tree sort [33] is given in appendix. The upper bound on computation for the modified algorithm can be shown to be $O(n^2)$ [29].

2.3 MINIMUM PATH SPANNING TREE

Theorem 2.2. A spanning tree T is a minimum path spanning tree of graph G if and only if for every subtree $S \subset T$ there is in T an edge contained in the smallest path from the root to a vertex in G–S, among all the edges connecting a vertex in S to a vertex in G–S.

From Theorem 2.2, it is apparent that a change in the labeling procedure in Dijkstra's algorithm would be sufficient to yield minimum path spanning tree [12]. In the modified labeling procedure a vertex v_i in Set B is given temporary label denoting the weight of the smallest path from root to v_i, passing through an edge in Set 2. This change in the implementation of Dijkstra's MST algorithm can be made by replacing

the statement following the comment by

$$z \leftarrow \text{label}(i) + D(i,j).$$

Above modification requires one extra addition for every edge in Set 2 per iteration, thus requiring n^2 additions more than the Prim and Dijkstra algorithm described in Section 2.3. This modification, however, does not alter the upper bound on the order of computation or the storage requirement of the algorithm. When Dijkstra's minimum path spanning tree algorithm is applied to a digraph it yields a minimum path spanning arborescence [12] . (See Figure 2.7).

2.4 MINIMUM SPANNING ARBORESCENCE

From the description of MST algorithms in Section 2.1, it is obvious that Kruskal's algorithm cannot yield spanning arborescence for a digraph, whereas the Prim and Dijkstra algorithm will produce a spanning arborescence if a digraph is inputted to it. Applying the Prim and Dijkstra algorithm to weighted digraphs one expects the resulting arborescence to be minimum spanning arborescence. But it is not so because Theorem 2.1 does not hold in case of directed graphs. This can also be shown with the help of following counter-example.

The given weighted digraph G is represented in Figure 2.4. Then Figure 2.5 represents the spanning arborescence T_1 obtained by applying the Prim and Dijkstra MST algorithm and Figure 2.6 represents the spanning arborescence T_2 obtained by applying Dijkstra's MPT algorithm.

The underlined integers in Figures 2.5 and 2.6 represent the permanent labels assigned to the vertices. Figure 2.7 represents a minimum spanning arborescence T_3 for the same digraph.

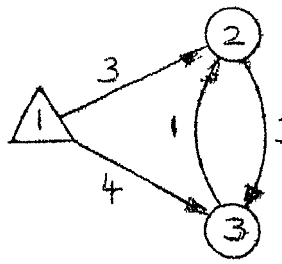


Figure 2.4: Digraph G.

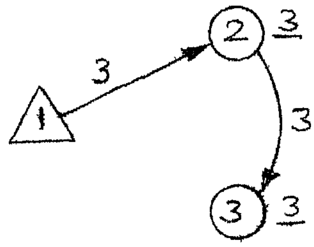


Figure 2.5: Spanning Arborescence T_1.

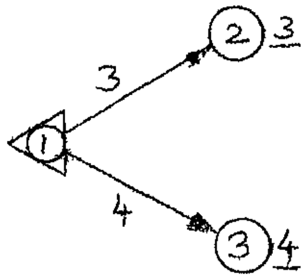


Figure 2.6: Spanning Arborescence T_2.

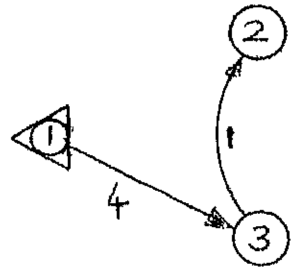


Figure 2.7: Spanning Arborescence T_3.

Observe that the weight of spanning arborescence in Figures 2.5 and 2.6 are $w[T_1] = 6$ and $w[T_2] = 7$, respectively, whereas the weight of spanning arborescence in Figure 2.7 is $w[T_3] = 5$.

2.5 EMPIRICAL RESULTS

The algorithms were coded in FORTRAN (listings are given in the appendix). The programs were tested on IBM 7044, with randomly generated

graphs of varying number of vertices and edges. The average execution time for each n and e was noted using a timer routine which gives time in units of $1/60^{th}$ of a second. The timer routine used takes a negligible time as compared to the quantum of measurement. The results are plotted from Figure 2.8 to Figure 2.11.

2.6 CONCLUSION

Speed is not the only measure in choosing an algorithm. Other important considerations are storage requirement, form of input, availability of algorithm and ease of implementation. In the light of this, the comparison between the various algorithms are done.

The Prim and Dijkstra algorithm is superior to the Kruskal algorithm in many of these respects. The computation time for nearly complete graphs is small compared to Kruskal and the algorithm is easy to implement. The storage requirements are quite small, specially when the graph is nearly complete and the edge weight is a simple computable function of the end vertices. Here the edge weight need not be stored at all, but can be computed when it is needed in Step 2 of the algorithm. With addition of tree sort the Prim and Dijkstra algorithm becomes much more useful for sparse graphs. But the algorithm looses its simplicity. Another advantage of the Prim and Dijkstra algorithm is its adaptability to minimum path spanning tree and minimum path spanning arborescence.

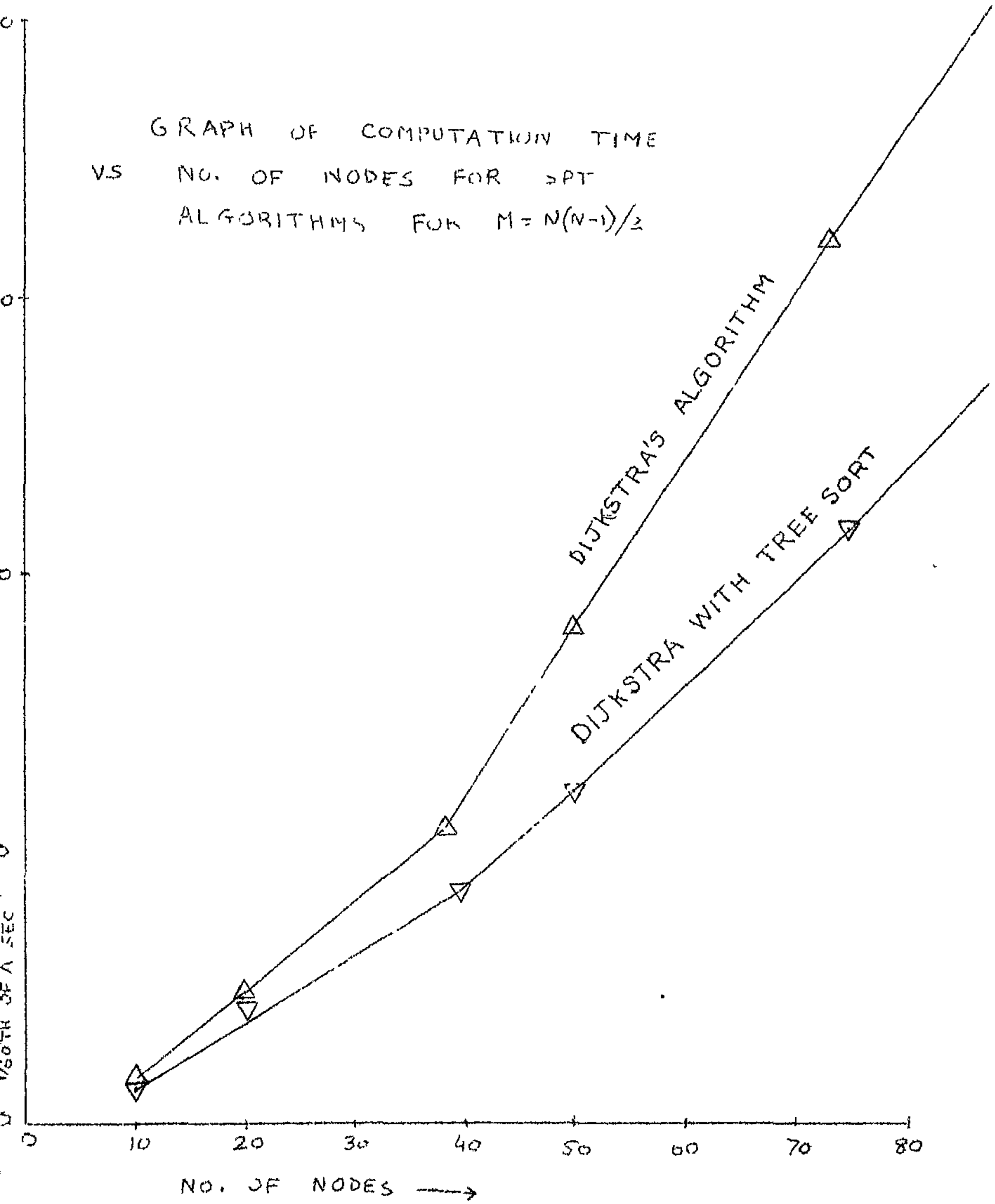

GRAPH OF COMPUTATION TIME
V.S NO. OF NODES FOR SPT
ALGORITHMS FOR M = N(N-1)/2
DIJKSTRA'S ALGORITHM
DIJKSTRA WITH TREE SORT
0
10
20
30
40
50
60
70
80
NO. OF NODES

FIG. 2.8

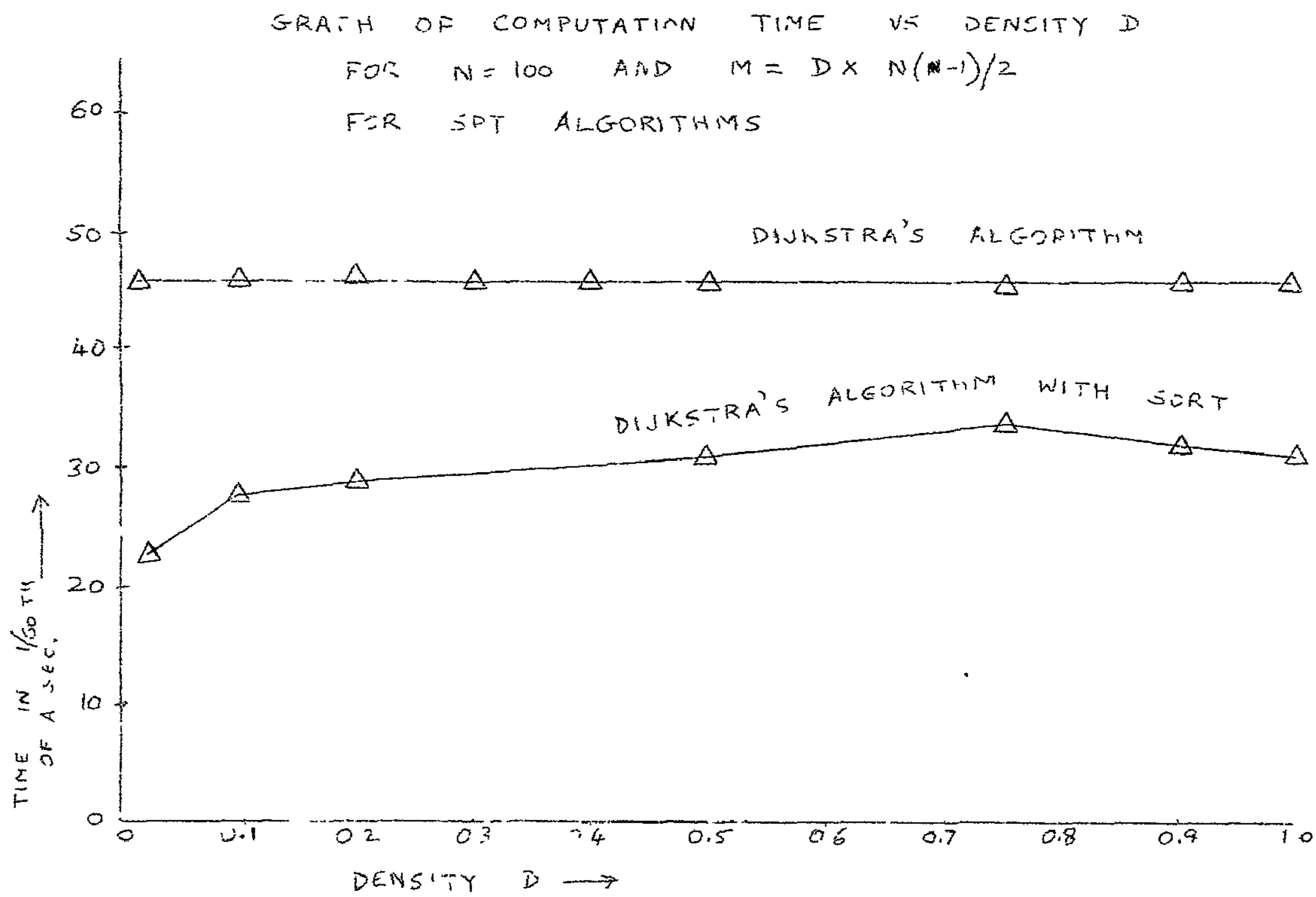

GRAPH OF COMPUTATION TIME VS DENSITY D
FOR N = 100 AND M = D X N(N-1)/2
FOR SPT ALGORITHMS
DIJKSTRA'S ALGORITHM
DIJKSTRA'S ALGORITHM WITH SORT
TIME IN 1/50 TH OF A SEC.
60
50
40
30
20
10
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
DENSITY D

FIG. 2.9

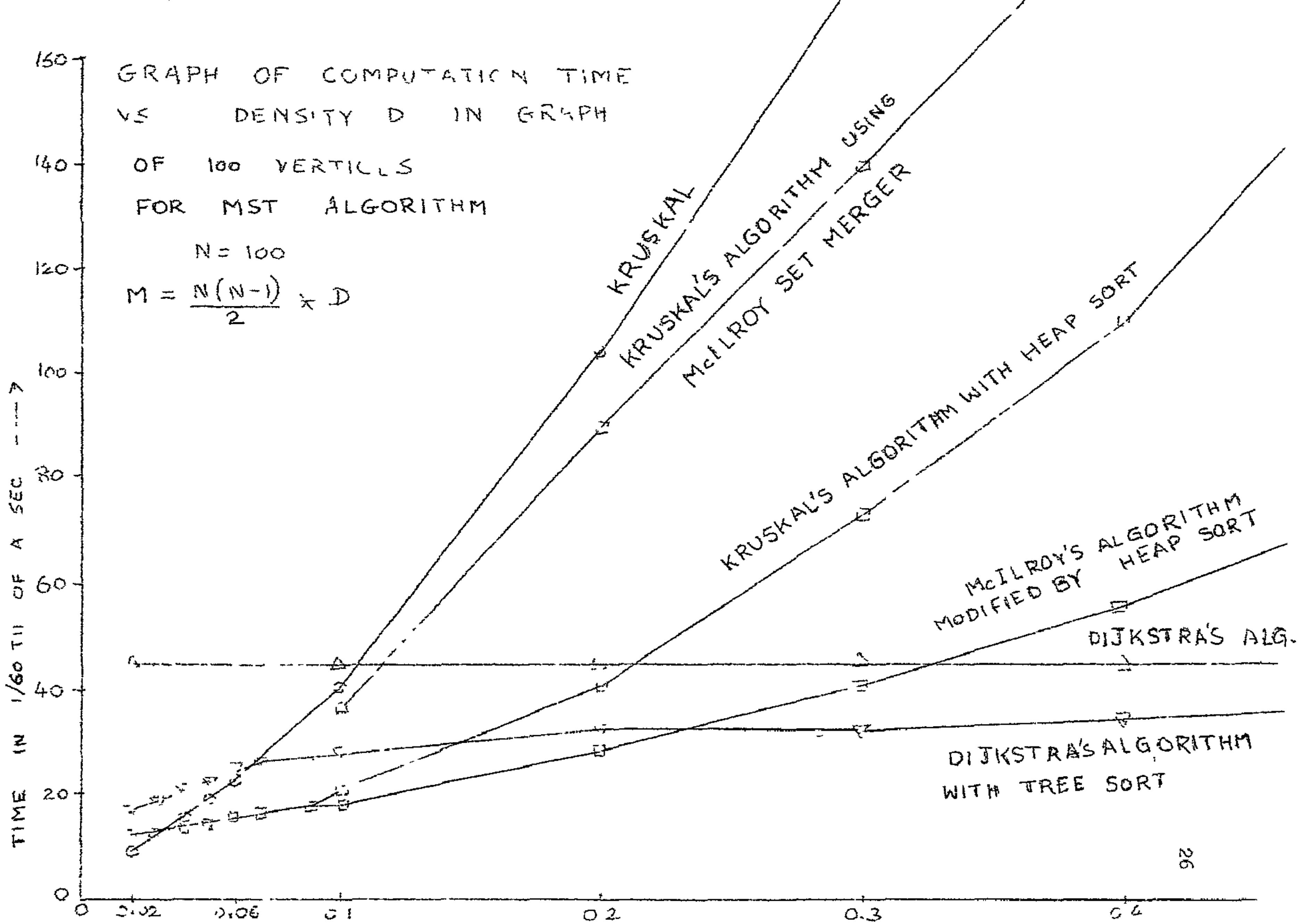

GRAPH OF COMPUTATION TIME
VS DENSITY D IN GRAPH
OF 100 VERTICLS
FOR MST ALGORITHM
N = 100
M = N(N-1)/2 * D
KRUSKAL
KRUSKAL'S ALGORITHM USING
McILROY SET MERGER
KRUSKAL'S ALGORITHM WITH HEAP SORT
McILROY'S ALGORITHM
MODIFIED BY HEAP SORT
DIJKSTRA'S ALG.
DIJKSTRA'S ALGORITHM
WITH TREE SORT
TIME IN 1/60 TH OF A SEC →
DENSITY →
0
20
40
60
80
100
120
140
150
0.02
0.06
0.1
0.2
0.3
0.4

FIG 2 10

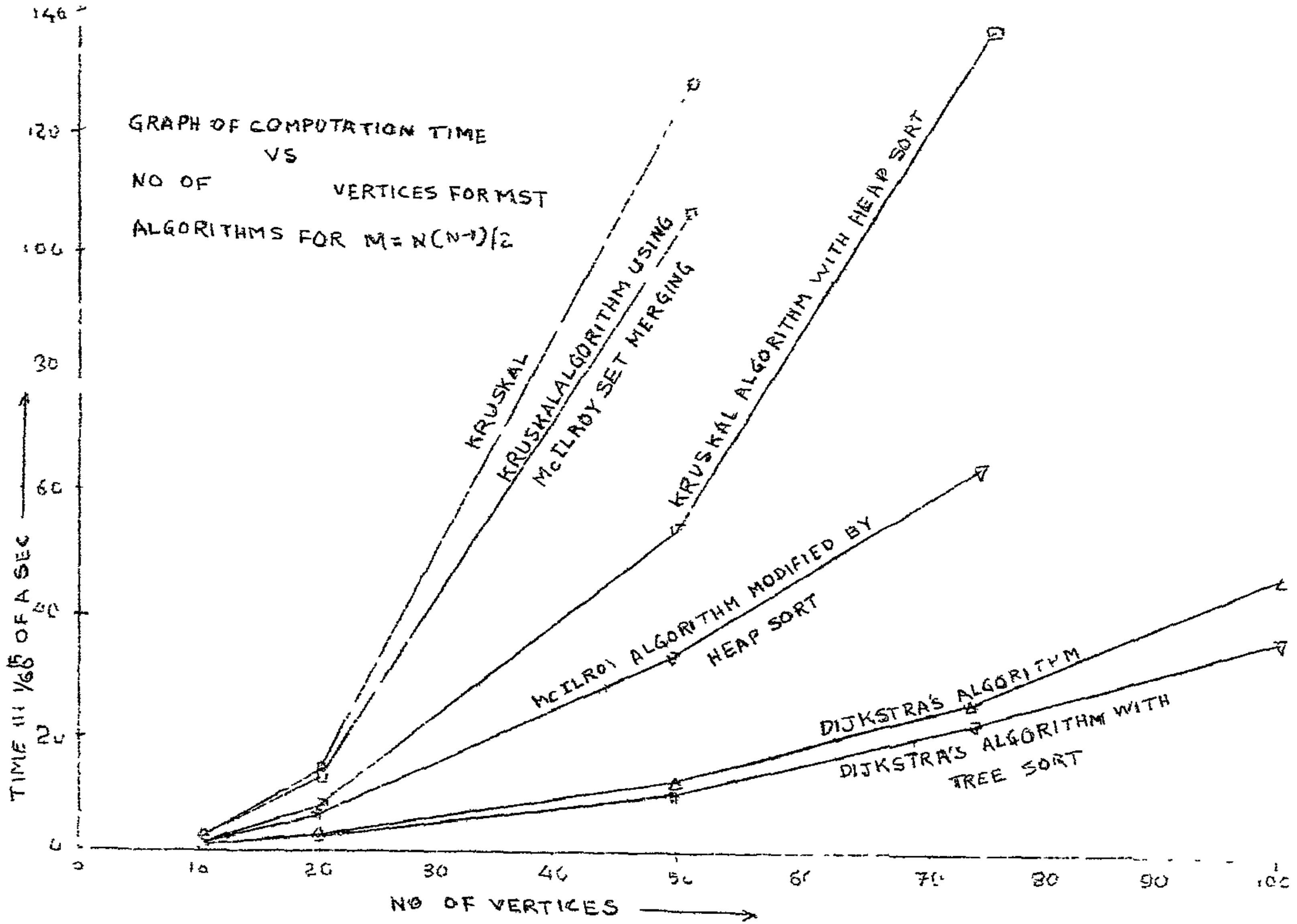

GRAPH OF COMPUTATION TIME
VS
NO OF VERTICES FOR MST
ALGORITHMS FOR M = N(N-1)/2
KRUSKAL
KRUSKAL ALGORITHM USING
McILROY SET MERGING
KRUSKAL ALGORITHM WITH HEAP SORT
McILROY ALGORITHM MODIFIED BY
HEAP SORT
DIJKSTRA'S ALGORITHM
DIJKSTRA'S ALGORITHM WITH
TREE SORT
TIME IN 1/60 OF A SEC
NO OF VERTICES
0
10
20
30
40
50
60
70
80
90
100
20
60
100
120
140

FIG 2 11

The Kruskal's algorithm as implemented by McIlroy is the best algorithm for generating minimum spanning tree or minimum spanning forest for sparse graphs. When used with heap sort, it performs well for density upto 0.3. The Kruskal algorithm requires a relatively large amount of memory because of the input representation of the graph. Also the implementation of the Kruskal algorithm is more complex than the Prim and Dijkstra algorithm.

CHAPTER 3

AN ALGORITHM FOR MINIMUM SPANNING ARBORESCENCE

In the previous chapter we have discussed the various algorithms for generating minimum spanning trees. We have also observed that these algorithms cannot be modified to obtain a minimum spanning arborescence (MSA) of a given digraph. An algorithm for generating an MSA of a given weighted digraph, its implementation and computational aspects are discussed in this chapter.

3.1 INTRODUCTION

Let G_o be a strongly connected weighted digraph. Then the minimum incidence subdigraph (MIS) G_o' is defined as a spanning subdigraph, consisting of a minimum weight edge incident into each vertex of the digraph G_o.

Let Reduced digraph G_1 of G_o (or G_i of G_{i-1}) be the digraph obtained by merging the vertices belonging to the cycles of G_o' into single vertices. Each such vertex is called a composite vertex. The process of merging is defined as follows: Let c be a cycle in G_o'. Then all edges incident out of any vertex $v_i \in V(c)$ are incident out of v_c. All edges incident into any vertex $v_i \in V(c)$, are incident into v_c. The weights of the edges incident into v_c are modified as follows:

$$w[\langle v_1, v_c \rangle] = w[\langle v_k, v_i \rangle] - w[\langle v_j, v_i \rangle] \qquad 3.1$$

where $v_1 \notin V(c)$ and $\langle v_j, v_i \rangle \in E(c)$.

By construction of MIS G_o',

$$w\left[\langle v_i, v_1\rangle\right] \geq w\left[\langle v_j, v_1\rangle\right] \qquad 3.2$$

Therefore the weights associated with the edges of reduced digraph G_1 are non-negative and if G_o is strongly connected then G_1 is also strongly connected.

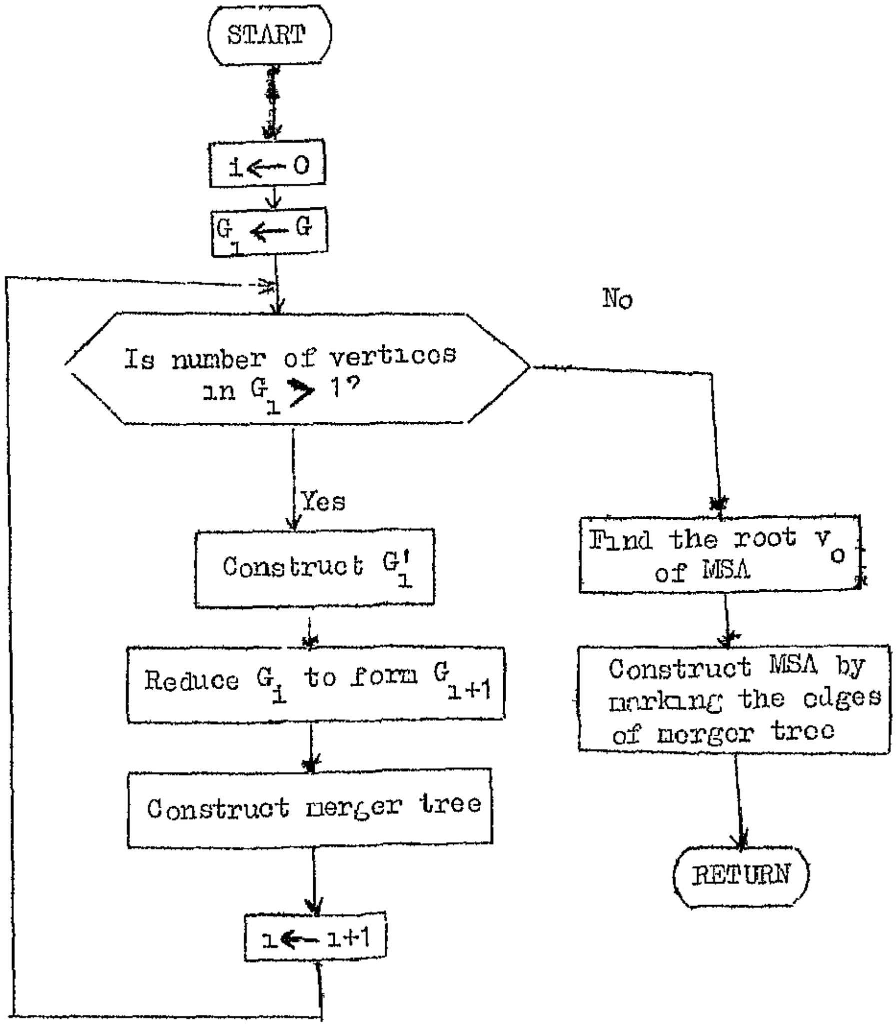
To find a minimum spanning arborescence of G_o, we construct a sequence of reduced digraphs $G_1, G_2, \ldots, G_{m+1}$, where G_{m+1} is a digraph with a single vertex. This process of reduction is represented by a Merger Tree M. The pendant vertices and the internal vertices of M correspond to the vertices of G_o and the composite vertices at different stages of reduction, respectively. Each composite vertex v_c (of G_i) in M has as its immediate successors, the vertices of G_{i-1} merged to form the composite vertex. The weight of the edge connecting v_c to its successor, say v_j, in M is $w\left[\langle v_i, v_j\rangle\right]$, where $\langle v_i, v_j\rangle \in E(c)$. The initial and terminal vertices of the edges corresponding to $\langle v_i, v_j\rangle$ in digraph G_o are also stored along with the vertex v_j of M.

Let an Arborescence Cover Digraph H be defined as spanning subdigraph of G_o where $E(H) = \bigcup_{j} \bigcup_{v_c \in G_j'} E(c)$ for $i = 1, 2, \ldots, m$. The arborescence cover digraph contains a minimum spanning arborescence.

The algorithm for generating an MSA of a digraph G_o can now be stated as follows: Given a strongly connected weighted digraph G_o, obtain the merger tree M. Choose the root v_o of the MSA as that pendant

vertex of merger tree which is farthest from the root of M. Mark the edges of M appropriately. The unmarked edges of M form the desired minimum spanning arborescence.

A block digram for the algorithm is given below.



3.2 MINIMUM INCIDENCE SUBDIGRAPH

Some properties of minimum incidence subdigraph are discussed here.

Property 3.1. Every vertex of a minimum incidence subdigraph G_i' has one and only one predecessor and $|E(G_i')| = |V(G_i')|$. As $E(G_i')$ contains exactly one edge incident into each vertex, the property 3.1 is immediate.

Theorem 3.1. Every connected component of a minimum incidence subdigraph G_i' has one and only one directed cycle.

Proof. Let G_s' be a connected component in G_i'. As every vertex in G_s' has a predecessor, G_s' must have a cycle. Again from property 3.1, $|E(G_s')| = |V(G_s')|$; therefore, G_s' cannot have more than one cycle.

Using the above properties of G_i' an algorithm for generating all cycles of G_i', in time proportional to n can be given (see Section 3.4).

3.3 DEVELOPMENT OF MSA ALGORITHM

Before proceeding with the algorithm, let us prove the following lemmas.

Lemma 3.1. Given any spanning arborescence T_i of G_i, there exists a spanning arborescence S_i of G_i with the same root as that of T_i, such that

(i) S_i contains $|E(c)|$ -1 edges from every cycle c of the MIS G_i'.

(ii) $w[S_i] \leqslant w[T_i]$.

<u>Proof</u>. If T_1 does not satisfy (i) then let T_1 contain at most $|E(c)| -2$ edges from cycle c of G_1'. Let $v_1 \in V(c)$ be farthest from the root (of T_1) in T_1, such that the edge $\langle v_1, v_1\rangle$ incident into v_1 in T_1 does not belong to $E(c)$. Let $\langle v_k, v_1\rangle \in E(c)$. Then v_k cannot be a successor of v_1 in T_1. For otherwise, let $v_1, v_2, \ldots, v_k$ be the path from v_1 to v_k in T_1.

By the choice of v_1,

$$\langle v_{k-1}, v_k\rangle \in E(c),$$

therefore, $v_k \in V(c)$.

By the same argument $\langle v_{k-2}, v_{k-1}\rangle \in E(c)$ and so on, until we come to the vertex v_1. Therefore, these k vertices in the path from v_1 to v_k in T_1 belong to the cycle c. As $\langle v_k, v_1\rangle$ is an edge in c and $v_1, v_2, \ldots v_k$ is a path in c from v_1 to v_k.

$$\{v_1, v_2, \ldots, v_k\} = V(c) \text{ and}$$

$$\{\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \ldots, \langle v_{k-1}, v_k\rangle, \langle v_k, v_1\rangle\} = E(c).$$

Therefore $|E(c)| = k$ or $|E(c)| -1 = k-1$, which contradicts the assumption.

Remove the sub arborescence rooted at v_1 and attach it to v_k through the edge $\langle v_k, v_1\rangle$. As v_k is not a successor of v_1 in T_1, the resulting digraph is also a spanning arborescence, say T_1' of G_1. Observe that T_1' has one more edge from cycle c than T_1 has. The edge set of T_1' can be obtained from that of T_1 by deleting the edge $\langle v_1, v_1\rangle$ and adding $\langle v_k, v_1\rangle$, i.e.,

$$E(T_i') = E(T_i) \cup \{\langle v_k, v_1\rangle\} - \{\langle v_l, v_1\rangle\} \qquad 3.3$$

By equation 3.1.

$$w\left[\langle v_k, v_1\rangle\right] \leqslant w\left[\langle v_l, v_1\rangle\right]$$

therefore,

$$\sum_{\forall \langle v_i, v_j\rangle \in E(T_i')} w\left[\langle v_i, v_j\rangle\right] \leqslant \sum_{\forall \langle v_i, v_j\rangle \in E(T_i)} w\left[\langle v_i, v_j\rangle\right]$$

or

$$w\left[T_i'\right] \leqslant w\left[T\right] \qquad 3.4$$

By repeating the above step at most $|E(c)| - 1$ times, we obtain the spanning arborescence T_i^* containing $|E(c)| - 1$ edges from cycle c of G_i'. As the cycles of MIS G_i' are all vertex disjoint, the above procedure can be applied independently to each cycle of G_i'. On doing so we get the spanning arborescence S_i of G_i containing $|E(c)| - 1$ edges from every cycle c of the MIS G_i'. It is immediate from the equation 3.4 that

$$w\left[S_i\right] \leqslant w\left[T_i\right] \qquad 3.5$$

Q.E.D.

Given a spanning arborescence T_i and the digraph G_i, the spanning arborescence S_i can be obtained by the above lemma. Next we shall see, how a spanning arborescence T_{i+1} for reduced digraph G_{i+1} of G_i can be obtained from T_i.

<u>Construction 3.1</u>. Let T_i be the given spanning arborescence of the digraph G_i. The corresponding spanning arborescence T_{i+1} of the reduced digraph G_{i+1} of G_i can be obtained from T_i as follows:

Step 1. Obtain S_i from T_i as in Lemma 3.1.

Step 2: Merge the vertices of S_i corresponding to each cycle of G_i'.

<u>Lemma 3.2</u>. Let T_i be the given spanning arborescence of G_i and T_{i+1} be the corresponding spanning arborescence of the reduced digraph G_{i+1}, then

$$w\left[T_{i+1}\right] \leqslant w\left[T_i\right] - \sum_{\forall c \in G_i'} w\left[c\right] + we_i$$

where $w[c]$ is the sum of the weights of the edges in cycle c and we_i is the weight of the edge incident into the root of T_i in G_i'.

<u>Proof</u>. Let $(v_1, v_2, \ldots, v, v_1)$ be the cycle c in the MIS G_i'. Note that they occure as a path say $v_1, v_2, \ldots, v_k$ in S_i (obtained from T_i in step 1 of construction 3.1). Let the path $v_1, v_2, \ldots, v_k$ be denoted by p. Let v_c be the composite vertex obtained by merging the vertices $v_1, v_2, \ldots, v_k$, then,

Case (i) If the set of vertices merged to form v_c contains the root of T_i (say v_1), then; let S_i' be an intermediate arborescence obtained from S_i by merging only the vertices of cycle c, then

$$\begin{aligned} E(S_i') &= E(S_i) - \bar{E}(p) \\ &= E(S_i) \cup \{\langle v_k, v_1\rangle\} - \bar{E}(p) \cup \{\langle v_k, v_1\rangle\} \\ &= E(S_i) \cup \{\langle v_k, v_i\rangle\} - E(c) \end{aligned}$$

Therefore,

$$w\left[S_i'\right] = w\left[S_i\right] + w\left[\langle v_k, v_1\rangle\right] - w[c]$$

As $w\left[\langle v_k, v_1\rangle\right]$ is denoted by we_i

$$w\left[S_i'\right] = w\left[S_i\right] + we_i - w[c] \qquad 3.6$$

Case 2. If the set of vertices merged to form v_c does not contain the root of S_i, then there is an edge $\langle v_1, v_c\rangle$ incident into v_c in T_{i+1}. Let the edge $\langle v_1, v_c\rangle$ in T_{i+1} correspond to the edge $\langle v_m, v_1\rangle$ in S_i. Then by equation 3.1

$$w\left[\langle v_1, v_c\rangle\right] = w\left[\langle v_m, v_1\rangle\right] - w\left[\langle v_k, v_1\rangle\right] \qquad 3.7$$

Now let S_i'' be an intermediate arborescence obtained from S_i by merging only the vertices of cycle c, then,

$$\begin{aligned} E(S_i'') &= E(S_i) \cup \left\{\langle v_1, v_c\rangle\right\} - \left\{\langle v_m, v_c\rangle\right\} - E(p) \\ &= E(S_i) \cup \left\{\langle v_1, v_c\rangle\right\} \cup \left\{\langle v_k, v_1\rangle\right\} - \left\{\langle v_m, v_c\rangle\right\} - E(c) \end{aligned}$$

Therefore, the weight of the arborescence S_i''

$$w\left[S_i''\right] = w\left[S_i\right] - w[c] + w\left[\langle v_1, v_c\rangle\right] + w\left[\langle v_k, v_1\rangle\right] - w\left[\langle v_m, v_1\rangle\right].$$

From Equation 3.7

$$w\left[S_i''\right] = w\left[S_i\right] - w[c] \qquad 3.8$$

As the cycles of MIS are all vertex disjoint, merging of the vertices corresponding to any cycle c of G_i' is independent of others. Therefore weight of T_{i+1} obtained by merging all the cycles of MIS G_i' can be written as follows:

$$w\left[T_{i+1}\right] = w\left[S_i\right] - \sum_{\forall c \in G_i'} w[c] + we_i \qquad 3.9$$

From Lemma 3.1

$$w[S_i] \leqslant w[T_i].$$

Therefore

$$w[T_{i+1}] \leqslant w[T_i] - \sum_{\forall c \in G_i'} w[c] + wc_i \qquad 3.10$$

Q.E.D.

The process of constructing T_{i+1} from T_i is continued till we arrive at a spanning arborescence T_{m+1} for the reduced digraph G_{m+1} containing only one vertex. Here note that T_{m+1} is a single vertex with no edges and so is S_{m+1}. Therefore $T_{m+1} = S_{m+1}$. Now we shall see how a given spanning arborescence S_i of G_i can be expanded to obtain a spanning arborescence S_{i+1} of $\bar{G}_{i+1}$.

Construction 3.2. Given any spanning arborescence S_i, of digraph G_i, the corresponding spanning arborescence S_{i-1} (containing $|E(c)| - 1$ edges from every cycle c of $\bar{G}'_{i-1}$, is constructed as follows. Let $v_1, v_2, \ldots, v_k$ be a cycle, say c, of $\bar{G}'_{i-1}$ merged to form v_c of G_i, then expand v_c as follows:

Step (i) If v_c is the root of S_i, then depending on which v_i we want to be the root of S_{i-1}, delete the edge $\langle v_{i-1}, v_i \rangle$ incident into the vertex v_i in cycle c.

Step (ii) If v_c is not the root S_i, then there is an edge $\langle v_1, v_c \rangle \in E(S_i)$ incident into v_c. Let $\langle v_1, v_c \rangle$ correspond to the edge $\langle v_m, v_1 \rangle$

of G_{i-1}, then delete the edge $\langle v_k, v_1 \rangle$ in cycle c.

Step (iii) Reassign the weight of edge $\langle v_1, v_c \rangle$ to correspond to the weight of edge $\langle v_m, v_1 \rangle$.

By applying the above steps to each composite vertex of S_i we get the spanning arborescence S_{i-1}.

Lemma 3.3. Let S_i be the given spanning arborescence of G_i, and S_{i-1} be the spanning arborescence of $\bar{G}_{i-1}$ obtained by expanding the composite vertices of S_i, then

$$w\{S_{i-1}\} = w\{S_i\} + \sum_{\forall c \in G'_{i-1}} w\{c\} - we_{i-1}$$

where we_{i-1} is the weight of the edge incident into the root of S_{i-1} in G'_{i-1}.

Proof. Let S'_{i-1} be an intermediate arborescence obtained by expanding the vertex v_c of the given spanning arborescence S_i. Let v_c be formed by merging vertices contained in cycle $(v_1, v_2, \ldots, v_k, v_1)$ say c of G'_{i-1}. Then

Case (i) : If v_c is the root of S_i then by step i of construction 3.2, the edge set of S'_{i-1} with v_1 as its root is given by

$$E(S'_{i-1}) = E(S_i) \cup E(c) - \{\langle v_k, v_1 \rangle\}.$$

Therefore

$$w\left[S'_{i-1}\right] = w\left[S_i\right] + w\left[c\right] - w\left[\langle v_k, v_1 \rangle\right]$$

or

$$w\left[S'_{i-1}\right] = w\left[S_i\right] + w\left[c\right] - we_{i-1}. \qquad 3.11$$

Case (ii) : If v_c is not the root vertex of S_i, then: Let the edge $\langle v_l, v_c\rangle$ incident on vertex v_c of S_i correspond to edge $\langle v_{li}, v_1\rangle$ of G_{i-1}. Then by step (ii) of construction 3.2, the edge set of S'_{i-1} is given as

$$E(S'_{i-1}) = E(S_i) \cup E(c) \cup \{\langle v_m, v_1\rangle\} - \{\langle v_l, v_c\rangle\} - \{\langle v_k, v_1\rangle\}.$$

Therefore

$$w\left[S'_{i-1}\right] = w\left[S_i\right] + w\left[c\right] + w\left[\langle v_m, v_1\rangle\right] - w\left[\langle v_l, v_c\rangle\right] - w\left[\langle v_k, v_1\rangle\right].$$

But from Equation 3.1

$$w\left[\langle v_l, v_c\rangle\right] = w\left[\langle v_m, v_1\rangle\right] - w\left[\langle v_k, v_1\rangle\right].$$

Therefore,

$$w\left[S'_{i-1}\right] = w\left[S_i\right] + w\left[c\right] \qquad 3.12$$

Expanding every composite vertex in the spanning arborescence S_i as above, we get the corresponding spanning arborescence $\bar{S}_{i-1}$. Then by equations 3.11 and 12 it is immediate that

$$w\left[S_{i-1}\right] = w\left[S_i\right] + \sum_{\forall c \in G'_{i-1}} w\left[c\right] - we_{i-1}$$

Q.E.D.

We will see in the following theorem that given any spanning arborescence T_o of G_o, the spanning arborescence S_o obtained by repeatedly merging the vertices of T_o to obtain T_{m+1} and then expanding it to obtain S_o, is the minimum weight spanning arborescence of G_o among all the arborescences with the same root as T_o.

Theorem 3.2. Given any spanning arborescence T_o, and arborescence cover digraph H of a strongly connected weighted digraph G_o, there exists a spanning arborescence S_o of G_o, with the same root as T_o, such that

$$(i)\ w[S_o] \leq w[T_o]$$

$$(ii)\ E(S_o) \subseteq E(H).$$

Proof. Let us prove the theorem using the principle of induction. Given T_o, construct a sequence of spanning arborescences $T_1, T_2, \ldots, T_{n+1}$ (using construction 3.1) where the spanning arborescence T_{n+1} consists of a single vertex, say v_{n+1} and no edges. It is obvious that root of T_o is contained in vertex v_{n+1}, of T_{n+1}, and the edge set of T_{n+1} (null set) is a subset of the edge set E_n of the arborescence cover subdigraph H of G_o.

Let $S_{n+1} = T_{n+1}$, then as a basis for induction hypothesis we have $w[S_{n+1}] = w[T_{n+1}] = 0$ and $E(S_{n+1}) = E(T_{n+1}) = \emptyset \subseteq E(H)$. Assume as the induction hypothesis, that S_i, a spanning arborescence of G_i, contains $|E(c)| -1$ edges from each cycle c of G_j' for $j = 1,2,\ldots,n$, the root of S_i contains the root of T_o,

$$w[S_i] \leq w[T_i] \text{ and } E(S_i) \subseteq E(H).$$

Obtain $\bar{S}_{i-1}$ from S_i as given in Construction 3.2 by choosing the root of S_{i-1}, such that the root of T_o is contained in it. $\bar{S}_{i-1}$ contains $|E(c)| -1$ edges from each cycle c of G_{i-1}' and S_i, by hypothesis,

contains $|E(c)|-1$ edges from every cycle c of G'_j for $j = i, i+1, \ldots, m$. Therefore S_{i-1} contains $|E(c)|-1$ edges from every cycle c of G'_j for $j = i-1, i, \ldots, m$. Again as all the edges added to S_i in forming S_{i-1} are contained in $E(H)$, $E(S_{i-1}) \subseteq E(H)$. Weight of the spanning arborescence S_{i-1}, by Lemma 3.3 is

$$w[S_{i-1}] = w[S_i] + \sum_{\forall c \in G'_{i-1}} w[c] - we_{i-1}$$

$$= w[S_{i+1}] + \sum_{\forall c \in G'_i} w[c] - we_i + \sum_{\forall c \in G'_{i-1}} w[c] - we_{i-1}$$

Proceeding similarly till we arrive at S_{m+1}, we have

$$w[S_{i-1}] = w[S_{m+1}] + \sum_{j,} \sum_{\forall c \in G'_j} w[c] - \sum_{j = i-1, i, \ldots, m.} we_j$$

$$j = i-1, i, \ldots, m.$$

$$= w[T_{m+1}] + \sum_{j,} \sum_{\forall c \in G'_j} w[c] - \sum_{j = i-1, i, \ldots, m} we_j$$

$$j = i-1, i, \ldots, m.$$

By repeated application of Lemma 3.2, we have

$$w[S_{i-1}] \leq w[T_m] + \sum_{j} \sum_{\forall c \in G'_j} w[c] - \sum_{j = i-1, 2, \ldots, m-1.} we_j$$

$$j = i-1, i, \ldots, m-1.$$

or

$$\leq w[T_i] + \sum_{\forall c \in G'_{i-1}} w[c] - we_{i-1}$$

therefore

$$w[S_{i-1}] \leqslant w[T_{i-1}] \qquad 3.14$$

Therefore by induction hypothesis

$$w[S_o] \leqslant w[T_o]$$

$$\text{and} \qquad E(S_o) \subseteq E(H).$$

Q.E.D.

Theorem 3.3. Given any strongly connected weighted digraph G_o and its arborescence cover digraph H, there exists a minimum spanning arborescence S_o of G_o, such that $E(S_o) \subseteq E(H)$.

Proof. Let T_o be a minimum spanning arborescence of G_o, and let S_o be the spanning arborescence obtained from T_o by Theorem 3.1. Then $w[S_o] \leqslant w[T_o]$ and $E(S_o) \subseteq E(H)$. But as T_o is an MSA $w[S_o] \nleq w[T_o]$, hence $w[S_o] = w[T_o]$. Therefore S_o is a minimum spanning arborescence of G_o contained in $E(H)$.

Q.E.D.

In Theorem 3.2 we observe that $T_{m+1} = G_{m+1}$ and the construction of G_{m+1} from G_o is independent of the given spanning arborescence T_o. Therefore, given a digraph G_o alone, it is possible to construct S_{m+1} and hence a minimum weight arborescence S_o with the given root. Then it is immediate that the minimum spanning arborescence can be obtained by expanding the reduced digraph $G_{m+1} = S_{m+1}$ for every vertex $v_i \in V(G_o)$, as the root of S_o, and choosing the minimum weight arborescence among them.

This would require us to apply the expansion procedure n times to reduced di graph G_{m+1}. This can be avoided if the choice of the root which will yield the MSA can be made a priori. The choice of the optimal root vertex can be made with the help of the merger tree M. The procedure for the construction of merger tree, the spanning arborescence S_o from the merger tree, and the choice of optimal root are described below.

Construction 3.3. Let the merger tree be initialised to contain n vertices corresponding to the vertices of G_o. Let v_c be a composite vertex of G_i formed by merging the vertices of the cycle $v_1, v_2, \ldots, v_k, v_1$ say c, of G'_{i-1}. The merging of cycle c into vertex v_c is represented by introducing vertex v_c into the merger tree and connecting the vertices $v_1, v_2, \ldots, v_k$ to v_c with edges $\langle v_c, v_1\rangle, \langle v_c, v_2\rangle, \ldots, \langle v_c, v_k\rangle$, where $\langle v_c, v_i\rangle$ corresponds to the edge $\langle v_{i-1}, v_i\rangle$. To facilitate the expansion of S_{m+1} to obtain S_o, the initial and terminal vertices of edges $\langle v_{i-1}, v_i\rangle$ in G_o are also stored with vertex v_i of M.

The expansion of G_{m+1} to obtain S_o as given by Theorem 3.2 can be implemented in terms of the merger tree by the marking procedure as follows.

Construction 3.4

Step (i): Let v_{m+1} be the root of the merger tree, then mark the edges in path from v_{m+1} to v_o, the root of s_o.

Step (ii) : If v_c is not the root of the merger tree, and the edge connecting v_c, to its predecessor $\langle v_{c+1}, v_c \rangle$ (corresponding to edge $\langle v_i, v_k \rangle$ is not marked, then mark the edge in the path from v_c to v_k in the merger tree.

Applying the above procedure breadth first to every vertex of merger tree we obtain a marking on the merger tree. The edges of G_o corresponding to the unmarked edges of merger tree represent the spanning arborescence So, with the given root.

The choice of the optimal root for S_o can be made using the following theorem.

Theorem 3.4. Let S_o, with root v_o, be a spanning arborescence of G_o containing $|E(c)|-1$ edges from every cycle c of G_j' for $j=1,2,\ldots,m$. Let M be the merger tree obtained from G_o and let P denote the path $v_{m+1}, v_m, \ldots, v_o$ from the root of the merger tree to v_o, then

$$w[S_o] = w[M] - w[P].$$

i.e., the weight of the spanning arborescence so obtained by marking the edges of M is equal to the weight of the merger tree minus the weight of the edges in the path from root of the merger tree to v_o, the root of the spanning arborescence s_o.

Proof: From Equation 3.13 we have

$$w[s_o] = w[S_{m+1}] + \sum_j \sum_{\forall\sigma \in G_j'} w(c) - \sum_{j=0,1,\ldots,m} we_j^1$$

$$j = 0,1,\ldots,m$$

As S_{m+1} has no edges $w[S_{m+1}] = 0$, therefore

$$w[S_o] = \sum_{j} \sum_{v_c \in G'_j} w[c] - \sum_{j = 0,1,\ldots,m} we_j \qquad 3.15$$

$$j = 0,1,\ldots, m$$

From the construction 3.3, we have

$$w[M] = \sum_{j} \sum_{v_c \in G'_j} w[c] \qquad 3.16$$

$$j = 0,1, \ldots, m$$

we_i denotes the weight of the edge incident on v_i, the root of S_i in G'_i. Observe that the edge incident on V_i in merger tree also has the same weight, therefore,

$$\sum_{j} we_j = w[p] \qquad 3.17$$

$$j = 0,1,\ldots, m$$

By Equations 3.15, 16 and 17

$$w[S_o] = w[M] - w[p].$$

Q.E.D.

Weight of the merger tree w M is a constant for a given digraph G_o. Therefore, $w[S_o]$ is minimum for that choice of root v_o for which $w[p]$ is maximum. Thus the optimal choice for root is that pendant vertex of merger tree which is farthest from the root of the merger tree.

3.4 AN MSA IMPLEMENTATION

An implementation for the construction of merger tree (Phase I), the optimal choice of root vertex (Phase II) and the marking procedure (Phase III) for obtaining an MSA of given digraph $G = G_o$ is given below.

MSA ALGORITHM

Integer Array: Mat(n,n), Mat i(m,n), Inod(n), Pred(n), Label(n), Stack(2n),
Mtree(2n,5), Avl(2n,2)

Integer: n, nmax, nov, cmax, cmin, min, imin, iroot, idmy, id, naul,
Paul, ipaul, troot, aroot, iroot, i,j, stack p, top,
vertex.

value : n

c: Mat represents the digraph G_o in input weight matrix form, Mati contains the initial vertices of every edge of G_1 corresponding to digraph G_o and Matj the terminal vertices. Inod gives a mapping between the rows and columns, and the vertex of G_1 it represents. Pred contains the predecessor list representation of the MIS and Mtree and merger tree. Avl is a list which contains the successor list of the internal vertices of merger tree.

N is the number of vertices in digraph G_o, nmax the maximum number in merger tree at any given time and nov the number of vertices in G_1;

```
begin: Initialization
 Paul ←1
 Naul← 500.
 for i = 1 step 1 until naul do
  begin:
   Avl(i,1) ← 0
   Avl(i,2) ← i
  end;
C: Set initial and terminal vertices matrix and initialize Inod and label.
 for i = 1 step 1 until n do
  begin:
   Inod(i) ← i
   Label(i) ← 0
   For j = 1 step 1 until n do
    begin:
     Mat i(i,j) ← i
     Mat j(j,i) ← i
    end;
  end;
C : Find the minimum entry in each column and subtract it from the column.
for i = 1 step 1 until n do
 begin:
  min ← ∞
  for j = 1 step 1 until n do
   begin:
    if Mat (j,i) < min then do
     begin:
      min ← Mat(i,1)
      imin ← j
     end;
   end;
   for j = 1 step 1 until n do; Mat(j,i) ←Mat(j,i) - min
C: Initialize merger tree
   Mtree(i,1) ← -1
   Mtree (i,3) ← Mat i(imn i)
   Mtrec(i,4) ← Matj(imin,j
   Mtree(i,5) ← min
  C: Form the predecessor list for MIS G'o.
     Pred(i) ← imin
   end;
  nov ← n
  nmax ← n
end; Initialization.
```

```
C : Phase I; construction of Merger tree M
C : Part I
    begin:
     while  nov   1 do
      begin:  Generation of cycles of MIS
       for vertex = 1 step 1 until n do
        begin : block 1
         top ← vertex; stack p ← 0
         if Lable(top) = 0 then do
          begin: block 1
           stackp ← stackp + 1
           stack (stackp) ← top
C: Label (top) = 0 indicates that the vertex is encountered for the
   first time.  When Lable Top ,≠ 0 implies that the vertex has appeared
   for the second time in the path i.e. a cycle is identified.
            while Lable (Top) .. 0 do
             begin:
              Label (top) ← stackp
              stackp ← stackp + 1
              stack (stackp) ← Pred(top)
              top ← stack (stackp)
             end;
           end; block 1
          if Label (top) ≯0 then
           begin : block 2
            cmin ← Label (top)
            cmax ← Stackp-1
            irow ← stack (cmin)
            Call MERGE (Mat, Mati, Matj, Pred, Label, Inod, Mtree,n,nov,
                        cmax, cmin, stack)
           end; block 2
           else:
            begin: block 3
             while stackp ≯ 0 do
              begin:
               top ← stack (stackp)
               Label (top) ← -1
               stackp ← stackp - 1
              end;
           end; block 3
         end;
      end;
```

```
  for i = 1 step 1 until n do
   begin:
    If Inod > 0 then label (i) ← 0
   end;
 end; Part 1

C: Part 2

   Procedure MERGE ( Mat, Mati, Matj, Pred, Label, Mtree, Avl, n, nov, cmin,
   cmax, stack)
   Integer Array. Mat(n,n), Mat i(n,n), Matj  (n,n), Pred(n), Label(n_j),
   Mtree(n,5), Avl(2n,2), stack(n)
   Integers: nov, cmax, cmin, nmax, idmy, min, imin,     .  pavl, ipavl,
   i,j, stackp, top
   Integer value: n

begin:

c: Merge the rows
 for i = 1 step 1 until n do
  begin: block 1
   min ← ∞
   imin ← irow
   for j = cmin step 1 until cmax do
    begin:
     idmy ← stack(i)
    if Mat  (i,idmy) < min then do
     begin:
      Imin ← idmy
      min ← Mat(i,idmy)
     end;
    end;
   Mat(i,irow) ← min
   Matj(i,irow) ← Matj(i,imin)
  end; block 1

C: Merge the columns

  for j = 1 step 1 until n do
   begin: block 2
    min ← ∞
    imin ← irow
    for i = cmin step 1 until cmax do
     begin:
      idmy ← stacki (i)
      if Mat (idmy,j) < min then do
       begin:
        min ← Mat(idmy,j)
        imin ← idmy
```

```
C: Modify the predecessor list
   If imin = 0 and Pred(j) = idmy then Pred(j)<-irow
  end;
end;
Mat(irow,j) <- min
Mati(irow,j) <- Mat i(irow,j)
Matj(irow) <- Matj(irow,j)
end;
Mat(irow,irow) <- ∞
nmax <- nmax + 1
nov <- nov - cmax + cmin
C: Form merger tree
  for i = cmin step 1 until cmax do
   begin:
    id <- stack(i)
    idmy <- Inod(id)
    Avl(pavl,1) <- idmy
    Mtree(idmy,2) <- nmax
    ipavl <- pavl
    pavl <- avl(pavl,2)
   end; block 2
  Avl(ipavl,2) <- 0
  Inod(irow) <- nmax
C: Find the minimum entry in the merged column
  min <- ∞
  Imin <- irow
  For i = 1 step 1 until n do
   begin: block 3
    if Inod(i) > 0 and Mat(i,irow) < min then do
     begin:
      min <- Mat(i,irow)
      imin <- i
     end;
    end; block 3
C: Subtract the minimum entry from the column
   for i = 1 step 1 until n do
    begin: block 4
     if Inod(i) > 0 then Mat (i,irow) <- Mat(i,irow) - min.
    end; block 4
```

```
C: Assign the weight and tag to the edge incident on nmax in Merger tree.
   Mtree (nmax,3) ← Mati(imin, irow)
   Mtree(nmax,4) ←Matj(imin,irow)
   Mtree(nmax,5) ← min
C: Assign predecessor to n max in G'_{i+1}
   Pred(irow) ← imin
   label(irow) ← -1
  end;
end; Part II.
C: Find the optimal root vertex:
C: Phase II
begin:
 troot ← nmax
 stackp ← 1
 stack(stackp) ← troot
 Mtree(troot,5) ← 0
C: Find path length from troot to all terminal vertices; label vertices
   of Mtree:

   while stackp ≠ 0 do
    begin: block 1
     iptr ← Mtree(troot,1)
     while iptr ≠ 0 do
      begin:
       idmy ← Avl(iptr,1)
       stackp ← stackp + 1
       stack(stackp) ←idmy
       Mtree(idmy) ← Mtree(idmy,5)+Mtree(troot,5)
       iptr ← Avl(iptr,2)
      end;
      troot ← stack (stackp)
      stackp ← stackp - 1
     end; block 1
C: Pick the vertex with largest path length (label) of Mtree
   for i = 1 step 1 until n do
    begin: block 2
     if Mtree(i,5) > min then do
      begin:
       min ← Mtree(i,5)
       aroot ← i
      end;
    end; block 2
   end; Phase II
```

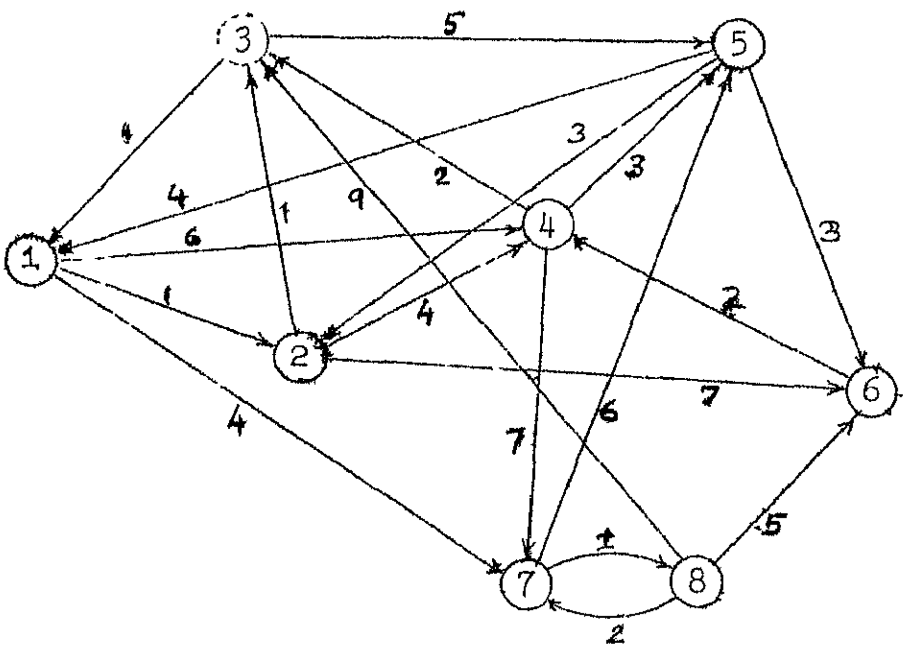
```
C: Merger Tree marking
C: Phase III
   begin:
    iroot ← aroot
    troot ← max
    stackp ← 1
    stack (stackp) ← troot
C: Mark the edges in path from troot to iroot
   while iroot < troot do
    begin:
     Mtree(iroot,5) ← -1
     iroot ← Mtree(iroot,2)
    end;
   while stackp > 0 do
    begin
     troot ← stack(stackp)
     iptr ← Mtree(troot,1)
     while  iptr > 0 do
   C: visit the next successor
      begin:
       idmy ← Avl(iptr,1)
       Mtree(troot,1) ← Avl(iptr,2)
       troot ← idmy
       stackp ← stackp + 1
       stack(stackp) ← idmy
       if Mtree(idmy, 5) ≥ 0 then
        begin:
         iroot ← Mtree(idmy,4)
         while iroot < troot do
          begin:
           Mtree(iroot,5) ← -1
           iroot ← Mtree(iroot,2)
          end;
        end;
      end;
     stackp ← stackp-1
    end;
   end; phase III
```

C: Store the unmarked edges in F and H arrays

```
begin. j <- 0
 for i = 1 step 1 until nmax do
  begin:
   if Mtree(1,5) > 0 then do
    begin: j <- j+1
           F(j) <- Mtree(i,2); H(j) <- Mtree(i,3)
    end;
  end;
end;
```

An Example

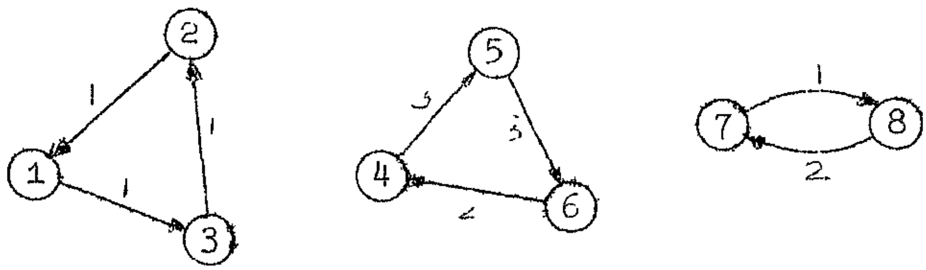
The input digraph G_o, initial merger tree and minimum incidence subdigraph G_o' are shown in Figure 3.1.



Input Digraph G_o

Figure 3.1

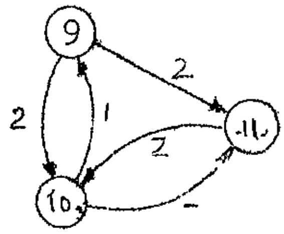
Initial merger tree



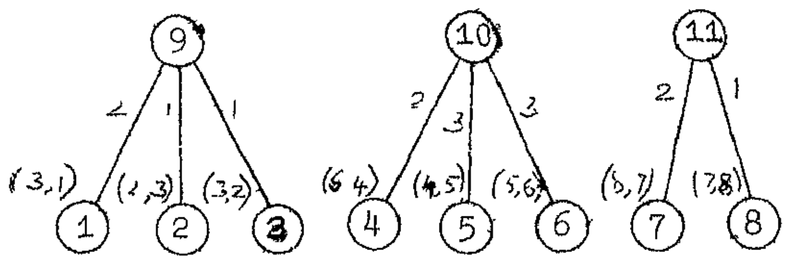
Composite vertex 9 Composite vertex 10 Composite vertex 11

Minimum Incidence Subdigraph G_o'

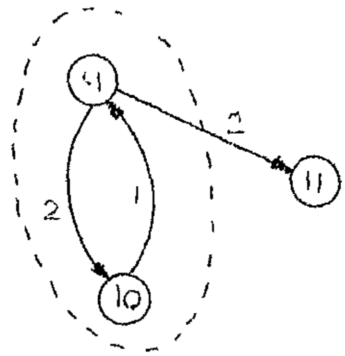
At the end of first iteration G_1, merger tree and minimum incidence subdigraph G_1' are shown in Figure 3.2. .



Reduced Digraph G_1



Merger Tree

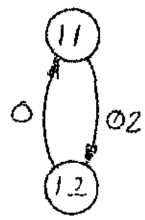


Composite vertex 12

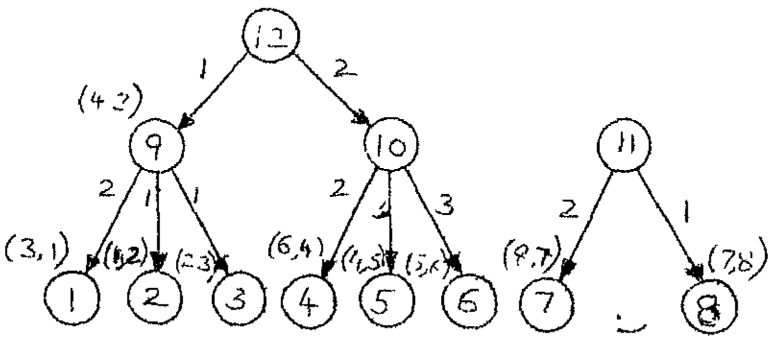
Minimum Incidence Subdigraph G_1'.

Figure 3.2 -

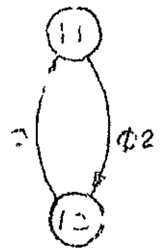
At the end of second iteration reduced digraph G_2, merger tree and minimum incidence subdigraph G_2' are shown in Figure 3.3.



Reduced Digraph G_2



Merger Tree



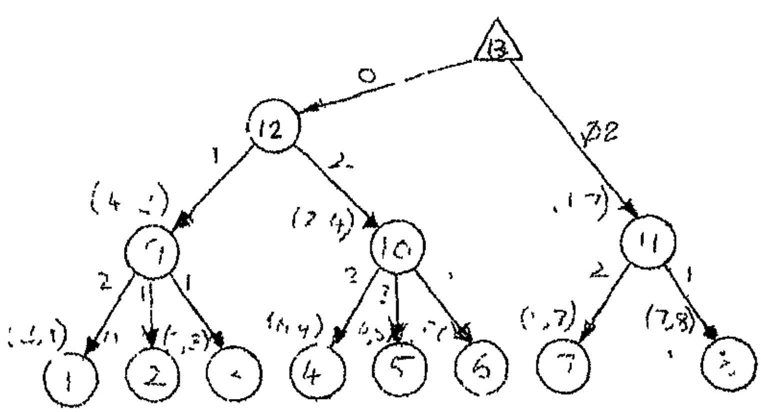
Composite vertex 13

Minimum Incidence Subgraph G_2'

Figure 3.3.

At the end of third iteration reduced digraph G_3 and merger tree M is shown in Figure 3.4.

Reduced Digraph G_3



The Merger Tree

Figure 3.4.

Phase 1 of execution is completed. Next we make the optimal choice of the root as shown in Figure 3.5 All underlined integers represent the weight of the path from the root to the corresponding vertices.

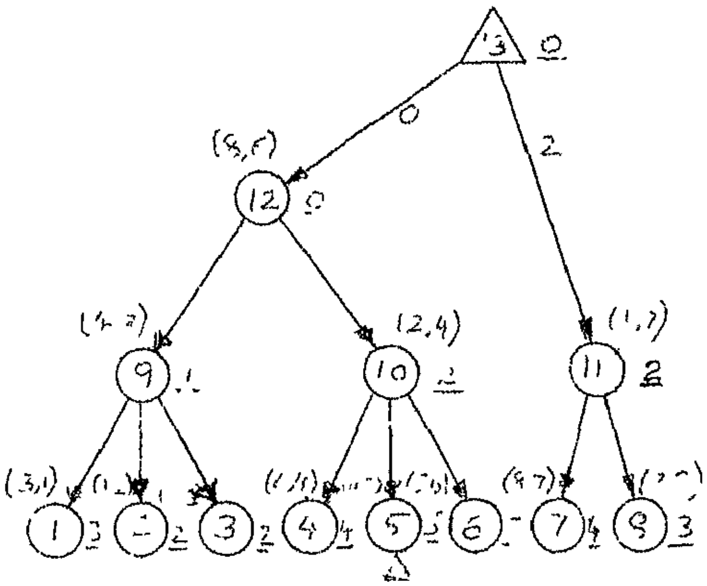
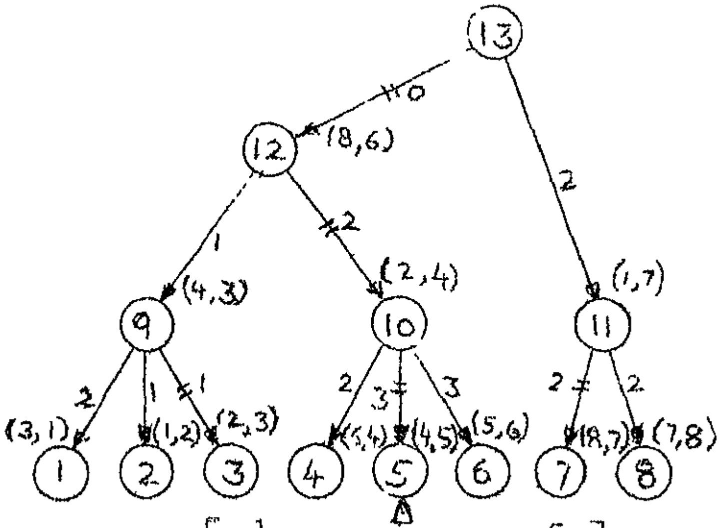


Figure 3.5

The marking as obtained by Phase III is shown in Figure 3.6 . The edges are marked by and the path P from root of M to root S_o is shown by double lines.



$W[M] = 20$ $\qquad$ $W[P] = 5$

Figure 3.6

The minimum spanning arborescence S_o formed by unmarked edges of merger tree M is shown in Figure 3.7 Note that $W[S_o] = W[M] - W[P] = 15$.

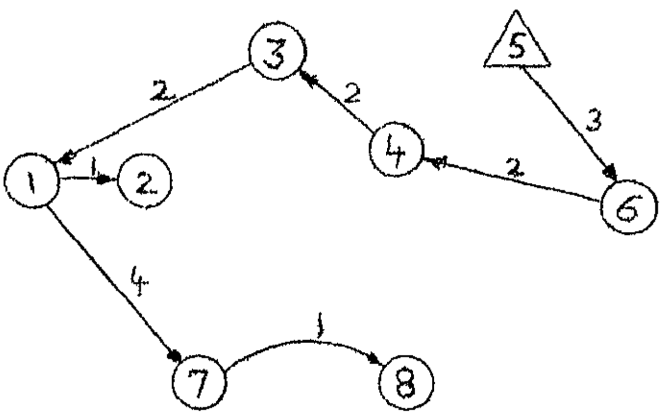


Figure 3.7

3.5 COMPLEXITY STUDIES

An upper bound on the order of computation required for the implementation of the minimum spanning arborescence algorithm is obtained here.

From a cursory look at the algorithm it is apparent that the upper bound on the computational complexity of the algorithm is determined by Phase I of the implementation, while Phase II and Phase III require much less computation. Therefore we would devote a major portion of this section in analysing Phase I.

For simplicity of analysis, Phase I has been split into two parts. Part 1 generates cycles of the given MIS and Part 2 merges vertices of the cycles generated in Part 1.

As the number of vertices or edges in any MIS G_i' does not exceed n, block 1 and block 3 of Part 1 require $O(n)$ computation per iteration. Block 2 is executed once for every cycle generated in Part 1. Postponing the analysis of MERGE for the present we analyse the remaining portions of Phase I.

The Phase I of the algorithm is executed a variable number of times depending on the process of merging. Each iteration represents generation of G_{i+1}' from G_i'. Therefore, the number of times Phase I is executed is equal to the number of vertices in the longest path from the root of merger tree to a pendant vertex. It is apparent that a merger tree (out-degree of every internal vertex being two

or more) with n pendant vertices will have at most n levels (as shown in Figure 3.8.). Therefore, Phase I may be executed at most n-1 times.

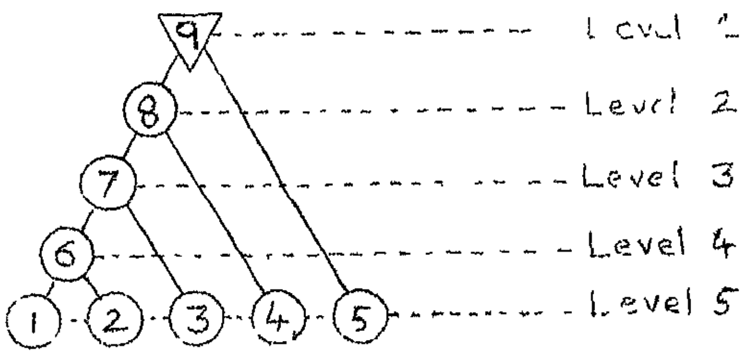


Figure 3.8.

Hence computation time for Part 1 of Phase I is bounded by $O(n^2)$.

As stated above block 2 of Part 1 is executed once for each cycle of MIS. As every cycle of MIS is represented in the merger tree by an internal (composite) vertex, block 2 is executed once for every internal vertex of the merger tree. The merger tree with n pendant vertices can have atmost n-1 internal vertices, the maximum being attained when every vertex in the tree has exactly two successors as shown in Figures 3.8 & and 3.9.

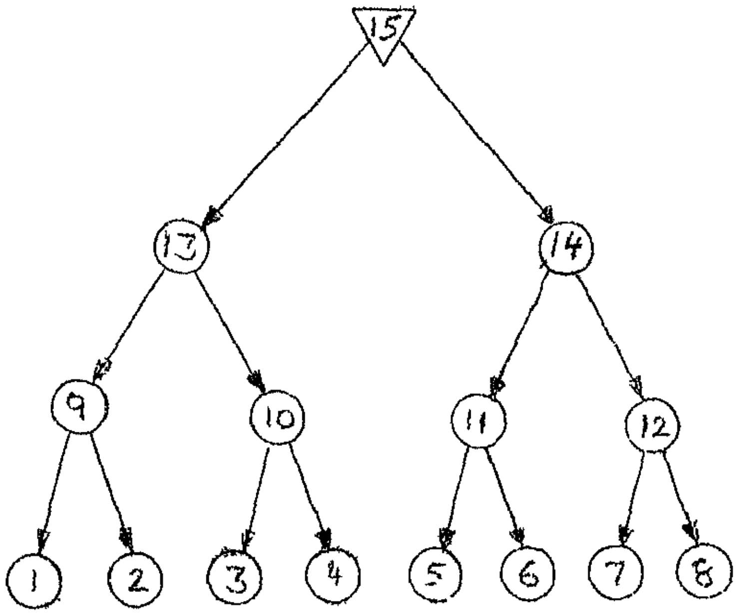


Figure 3.9

Therefore, in the worst case block 2 is executed n-1 times. Now let us analyse the complexity of Part II, the MERGE procedure, independently. Here we see that all four blocks are executed once every time the MERGE procedure is called. Block 1 and Block 2 require $n \times |E(c)|$ computation for each call to the procedure. Hence the total computation time in block 1 and 2 is proportional to:

$$\sum_{\substack{j \\ j = 0,1,\ldots,n}} \sum_{\forall c \in G'_j} n * |L(c)| = n \sum_{\substack{j \\ j = 0,1,\ldots,n}} \sum_{\forall c \in G'_j} |E(c)|$$

As can be seen from the construction 3.3, the summation represents the number of edges in the merger tree. The maximum number of edges possible in a merger tree is 2n-2 (see Figure 3.9).
Therefore, the total computation time for block 1 and 2 of MERGE is at most n(2n-2).

Block 3 and block 4 each requires computation of the order of n, and are executed once for every call to the procedure. Therefore total computation time is bounded by $n(n-1)$.

Hence the procedure MERGE is bounded by $O(n^2)$. Thus the order of complexity of Phase I of the algorithm is $O(n^2)$.

Phase II of the algorithm is divided into two blocks. Block 1 is a tree traversing algorithm for the merger tree. The merger tree may have atmost 2n-1 vertices and 2n-2 edges, therefore, the block 1 requires computation of $O(n)$. Block 2 finds the vertex with maximum label among the n terminal vertices of the merger tree and hence requires computation of $O(n)$.

Marking procedure for the merger tree in Phase III is logically divided into two parts. First, tree traversing which requires computation of the order of n, and second, edge marking. The merger tree has atmost 2n-2 edges and atmost n-1 of these edges are marked. Marking an edge requires a constant amount of computation. Therefore, Phase III of the algorithm requires computation of $O(n)$.

Thus complexity of the algorithm is of the order of n^2. Storage requirement for the algorithm can also be seen to be of the order of n^2.

3.6 EMPIRICAL RESULTS AND CONCLUSIONS

Tests similar to that described in Chapter 2 were carried out for the above algorithm. The results are tabulated in Table 3.1.

n / D	10	20	30	40	50	60	70
0.5	7.0	21.39	44.69	74.3	115.4	161.2	216.6
0.75	6.8	22.0	43.2	75.7	113.5	157.4	212.2
0.9	6.8	20.7	42.4	77.3	117.7	161.2	207.1
1.0	7.2	20.7	45.1	71.9	116.3	161.1	220.2

Table 3.1

The variation of the computation time with the number of vertices, for complete graphs is plotted in Figure 3.10 From Figure 3.10' we observe that the average computation time required is of the order of $n^{1.95}$. On the average a digraph with 70 vertices requires about 4 seconds of time and a little over 15K words of memory (data) locations It can also be seen from the above table that the computational time required is independent of the number of edges. This is so because the input representation of the digraph is in the nxn weight matrix form. A list structure for representing the input digraph could also be used. But this would make the algorithm much more complex and would in fact result in deterioration in the average performance.

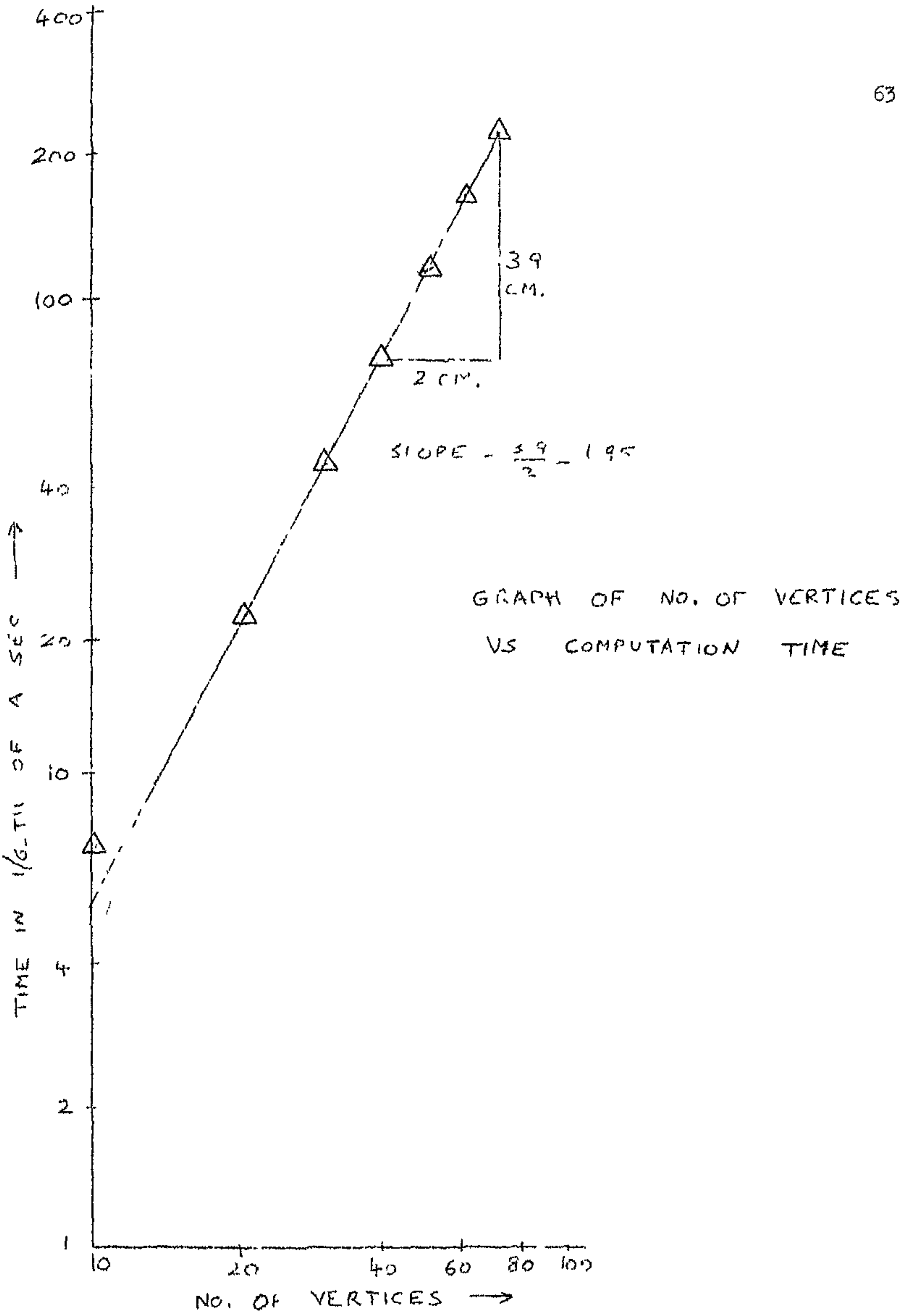

400
200
100
40
20
10
4
2
1
TIME IN 1/60 TH OF A SEC
39 CM.
2 CM.
SLOPE = 3.9/2 = 1.95
GRAPH OF NO. OF VERTICES VS COMPUTATION TIME
10
20
40
60
80
100
NO. OF VERTICES

FIG. 3.10

The above algorithm chooses the optimal root by itself. But some of the applications may require a minimum weight spanning arborescence with a given root. This could be obtained by deleting Phase II of the algorithm and initializing 'aroot' to the root vertex desired before the execution of Phase III. If minimum rooted arborescence with different root vertices are desired they can be obtained by first generating the merger tree (Phase I) and then repeating the marking procedure (Phase III) for every root vertex (corresponding to which minimum arborescence is desired).

CHAPTER 4

CONCLUSION

Of all the analysed algorithms for finding minimum spanning trees, the Prim-Dijkstra algorithm modified with tree sort is the best algorithm for nearly complete graphs. The algorithm is simple and elegant, and can be easily modified to obtain minimum path spanning tree and minimum path spanning arborescence.

Of the different implementations of Kruskal's algorithm discussed, McIlroy's implementation [37] using the set merging algorithm is the best. The McIlory's implementation modified with heap sort [42] gives best results for density upto 0.3. Also when the input graph is sparse or when the input graph is larger than the available memory space, Kruskal's algorithm is much faster than the Prim and Dijkstra algorithm.

4.1 APPLICATIONS AND FUTURE PROBLEMS

Some of the applications of minimum spanning trees were mentioned in Chapter 1. Those applications are discussed in this section.

The most obvious application of MST is in minimum connecting network problem [29]. The minimum connecting network problem can be used for various problems such as the minimum connecting road network [9], minimum length wiring [42], minimum cost communication and transportation networks [41] etc. The minimum connecting road network problems can be stated as follows: Suppose that we have to connect n cities v_1, v_2, ..., v_n

through a network of roads. The cost C_{ij} of building a direct road between v_i and v_j is given for all pairs of cities where roads can be built. The problem is then to find the least cost network that connects all n cities. It is immediately evident that this network is a minimum spanning tree.

A problem similar to that stated above is as follows. Suppose we have a warehouse at one city say v_1, and want to set up a commodity distribution system over n cities $v_1, v_2, \ldots, v_n$. The cost c_{ij} of transporting unit commodity between v_i and v_j is given for all pairs of cities which are directly connected to each other. The problem is to set up least cost commodity distribution system based at city v_1. Obviously the solution is a shortest path tree. This well-known problem in Operations Research is known as transportation problem [25].

Let us now assume that the cost incurred in traversing any edge is independent of the amount carried through it. In this case the solution is a minimum spanning tree.

Some problems in social science can be modeled in the form of weighted digraphs [24]. Consider one such hypothetical problem of constructing optimal command structure in a group of people, $v_1, v_2, \ldots, v_n$. Let an edge $\langle v_i, v_j \rangle$ with weight c_{ij} denote the resistance of v_i to obey v_j. Then the problem is to find a command structure in the group such that the orders given by an appointed leader are best followed. The solution to this problem is an MSA.

Spanning tree algorithms are used in solving various graph theory problems such as finding all fundamental circuits in a graph [9], travelling salesman problem [25], connectedness in graphs [26] etc. One interesting problem in digraphs is finding the minimum equivalent subdigraph of a digraph. A minimum equivalent subdigraph of a digraph G is a minimal subdigraph having the reachability property of G. This problem is closely related to the spanning tree problem. It can be easily seen that a spanning tree of a undirected graph G has the reachability property of G. But in the case of strongly connected digraphs this problem can be shown to be atleast of the complexity of Karp-Cook class [28]. A tight upper and lower bound on the weight of a minimum equivalent subdigraph of a weighted digraph and an approximate solution for it can be obtained with the help of MSA as shown below.

All known algorithms [11,38] for the above problems use search techniques. The search time for the algorithms can be considerably reduced if tight upper and lower bound on the weight of the solution can be found. It is easy to see that the weight of a minimum equivalent subdigraph is bounded below by the weight of the MSA. It is bounded from above by the sum of the weights of the minimum spanning arborescence and the minimum spanning arborescence on the digraph obtained from G by reversing the direction of all the edges in G.

This bound can be made tighter by first obtaining an MSA, replacing by zero the weights of the edges of the graph G that belong to MSA and then obtaining the second MSA. As a matter of fact an approximate solution to minimum equivalent sub-digraph is given by the union of the edges of the two minimum spanning arborescences. An exact algorithm for finding a solution to the above problem has been given by Moyles and Thompson [38] and an efficient algorithm for acyclic digraphs by Deo and Krishnamoorty [11].

REFERENCES

1. Auguston, J.G., and Minker, "An analysis of some graph-theoretic cluster techniques," J.ACM, Vol. 17, No. 4, October 1970, pp. 571-588. (Correction in J.ACM, Vol. 19, April 1972, pp. 244-247.

2. Berge, C., "Theory of graphs and its applications," John-Wiley and Sons, New York, 1962.

3. Boruvika, O., "On a minimal problem," Proce. Moravske Pridovedecke Spodeenosti, 3, 1926.

4. Chang, S.K., "The generation of minimal trees with stiner topology," J.ACM, Vol. 19, October 1972, pp. 699-711.

5. Chen, W., "Applied graph theory," North-Holland Pub. Co., Amsterdam, 1971.

6. Choquet, G., "Etude de certains rescan de routi," C.R.Ac.Sc. 206, 310, 1938.

7. Dahl, O.J., E.W. Dijkstra and C.A.R. Hoare, "Structured Programming," Academic Press, London and New York, 1972.

8. Dantzig, G., D.R. Fulkerson and S. Jonson, "Solution of a large traveling salesman problem," Operations Research, Vol. 2, November 1954, pp. 343-410.

9. Deo, N., "Graph theory with applications to engineering and computer science," Prentice-Hall, Englewood Cliffs, N.J., 1974.

10. Deo, N., "Breadth-and depth-first search in graph-theoretic algorithms," Tech. Report, TRCS-74-001, Indian Institute of Technology, Kanpur, January 1974.

11. Deo, N., and M.S. Krishnamoorty, "An algorithm for removing surplus edges from a directed graph," Submitted to J.ACM.

12. Dijkstra, E.W., "A note on two problems in connection with graphs," Numerisch Math., Vol. 1, 1959, pp. 269-271.

13. Dreyfus, S.E., "An appraisal of some shortest path algorithms," Operations Research, Vol. 17, No. 3, 1969, pp. 395-412.

14. Even, S., "Algorithmic Combinatorics," The Macmillan Co., New York, 1973.

15. Fischer, M.J., "Efficiency of equivalence algorithms," Complexity of Computer Computation, Miller, et al., eds., Plenum Press, New York, 1972, pp. 153-167.

16. Floyd, R.W., "Tree sort," Algorithm 113, ACM Collected Algorithms, August 1962.

17. Floyd, R.W., "Treesort," Algorithm 113, ACM Collected Algorithms, December 1964.

18. Floyd, R.W., "Algorithm 97, Shortest path," Comm. ACM, Vol. 5, 1962, pp. 345.

19. Frazer, W.D., "Analysis of combinatorial algorithms - A sample of current methodology," AFIPS

20. Gallar, B.A. and M.J. Fischer, "An improved equivalence algorithm," Comm. ACM, Vol. 7, 1964, pp. 310-303.

21. Garfinkel, R.S. and G.L. Nemhauser, "Integer Programming," John Wiley and Sons, New York, 1972.

22. Gilbert, E.N., and H.O. Pollack, "Steiner minimal trees," J.SIAM, Vol. 16, 1968, pp. 1-29.

23. Hadley, G., "Linear Programming," Addison-Wesley, Reading, Mass.

24. Harary, F., "Graph Theory," Addison-Wesley, Reading, Mass., 1969.

25. Harary, F., R.Z. Norman and D. Cartwrite, "Structural models : Introduction to the theory of directed graphs," Wiley, New York, 1955.

26. Held, M. and R.M. Karp, "The traveling salesman problem and minimal spanning trees," Mathematical Programming, Vol. 1, 1971, North-Holland Publishing Co., pp. 6-25.

27. Hopcroft, J. and R. Tarjan, "Efficient algorithm for graph manipulation," Comm. ACM, Vol. 16, 1973, pp. 372-376.

28. Hopcroft, J., and J.D. Ullman, "Set merging algorithms," SIAM J. of Comput. Vol. 2, December 1973, pp. 294-303.

29. Karp, R.M., "Reducability among combinatorial problems," Complexity of Computer Computations, Miller et al., eds., Plenum Press, New York, 1972, pp. 85-103.

30. Kirchhoff, G., "Uber die Auflosung der Gleichungen, auf welche mass bei dev untersuchungen der linearen verteilung Galvanishev stromo gefuhrt wird," Poggendorf Ann. Physik, Vol. 72, 1847 - 508.

English translation, IRE Trans. Circuit Theory, Vol. CT-5, March 1958, pp. 4-7.

31. Kershenbaum, A. and R. Van Slyke, "Computing minimum spanning trees efficiently," Proc. ACM 1972 Annual Conf., New York, 1972, pp. 518-527.

32. Knuth, D.E., "The art of computer programming : Vol. I -- Fundamental algorithms," Section 2.3.3 and 2.3.4, Addison-Wesley, Reading, Mass., 1968.

33. Knuth, D.E., "The art of computer programming : Vol. III-- Sorting and searching," Section5.2.3, Addison-Wesley, Reading, Mass., 1973.

34. Kruskal, J.B., "On the shortest spanning subtree of a graph and the traveling salesman problem," Proc. Amer. Math. Society, Vol. 7, 1956, pp. 48-50.

35. Liu, C.L., "Introduction to combinatorial mathematics," McGraw-Hill, New York, 1968.

36. Loberman, H. and Weinberger, "Formal procedures for connecting terminals with a minimal total wire length," J.ACM, Vol. 4, 1957, pp. 428-437.

37. McIlroy, M.D., "Algorithm 354 : Generator of spanning trees," Comm. ACM, Vol. 2, September 1969, pp. 511.

38. Moyles, D.M. and G.L. Thompson, "Algorithm for finding a minimum equivalent graph of a digraph," J.ACM, Vol. 16, 1969, pp. 455-460.

39. Obruce, A., "Algorithm 1, Mintree," Computer Bulletin, September 1964, pp. 67.

40. Peterson, M., Unpublished Report, Univ. of Warwick, Coventry, Great Britain.

41. Plane, D.R. and C. McMillan, "Discrete optimization : Integer programming and network analysis for management decisions," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

42. Prim, R.C., "Shortest connecting networks and some generalizations," Bell Systems Tech. J., November 1957, pp. 1389-1401.

43. Read, R.C., "Teaching graph theory to a computer," Recent progresses in combinatorics, Ed. by W.T. Tutte, Academic Press, New York, 1967.

44. Rosenstiechl, P., "L'Arbre minimum d'um graphe," Theory of Graphs, Ed. by P. Rosenstiechl, Gordon and Breach, New York, 1967.

45. Seshu, S. and M.B. Reed, "Linear graphs and electrical networks," Addison-Wesley Pub. Co., Reading, Mass., 1961.

46. Seppanen, J.K., "Spanning tree (H)," Algorithm 399, Comm. ACM, Vol. 13, October 1970, pp.

47. Tarjan, R.E., "Depth-first search and linear graph algorithms," SIAM J. of Computing, Vol. 1, No. 2, June 1972, pp. 146-160.

48. Tarjan, R.E., "On the efficiency of a good but not linear set union algorithm," Tech. Report 72-148, Dept. of Comp. Sc., Cornell Univ., Ithaca, New York, 1972.

49. Wagner, H.M., "*Principles of operations research*," Prentice-Hall, Englewood Cliffs, N.J., 1969.

50. Williams, J.W.J., "Heap sort," Algorithm 232, *ACM Collected Algorithms*, January 1964.

APPENDIX I. SEPPANENS IMPLEMENTATION OF KRUSKAL ALGORITHM

```
C
C
£IBFTC SPTRE
      SUBROUTINE SPTRE(F,H,N,E,EDGE,C,W)
C
C
C     SEPPANEN&S IMPLEMENTATION GROWS A SPANNING TREE T FOR
C     A GIVEN UNDIRECTED GRAPH OF N VERTICES AND E EDGES.
C     IF THE INPUT GRAPH IS DISCONNECTED A SPANNING FOREST
C     WILL BE GENERATED.
C     THE TWO TERMINAL VERTICES OF THE I&TH EDGE ARE STORED
C     IN F(I) AND H(I) AND ITS WEIGHT IS STORED IN W(I).
C     THE ARRAY EDGE(I) DENOTES THE COMPONENT WHICH CONTAINS
C     THE I&TH EDGE.
C     VERTEX(I) DENOTES THE COMPONENT WHICH CONTAINS VERTEX VI.
C
      INTEGER C,E,EDGE(3000),F(3000),H(3000),VERTX(3000),V1,V2,W(3000)
      DIMENSION NO(3000)
      DO 4 L = 1,N
    4 VERTX(L)=0
      DO 6 L = 1,E
      NO(L)=L
    6 EDGE(L)=0
      CALL SORT(E,W,NO)
      C=0
      M=0
      K=0
      KZ=0
   10 KZ=KZ+1
C
C     PICK AN EDGE
C
      K=NO(KZ)
      V1=F(K)
      I=VERTX(V1)
      IF(I)399,39,399
  399 V2=H(K)
      J=VERTX(V2)
      IF(J)37,36,37
   37 IF(I-J)21,50,18
C
C     VERTEX V1 IS IN COMPONENT I AND VERTEX V2 IS IN
C     COMPONENT J.
C     GRAFT THE TWO TREES CONTAINING V1 AND V2.
C
   18 IJI=J
      J=I
      I=IJI
   21 DO 26 L = 1,N
      IF(VERTX(L)-J)26,25,26
   25 VERTX(L)=I
```

```
      GO TO 26
   25 VERTX(L)=VERTX(L)-1
   26 CONTINUE
      DO 32 L=1,E
      IF(EDGE(L)-J)32,29,31
   29 EDGE(L)=I
      GO TO 32
   31 EDGE(L)=EDGE(L)-1
   32 CONTINUE
      C=C-1
      EDGE(K)=I
      GO TO 49
C
C     V1 IS IN A COMPONENT AND V2 IS NOT IN ANY COMPONENT.
C      ADD V2 TO THE COMPONENT.
C
   36 EDGE(K)=I
      VERTX(V2)=I
      GO TO 49
   39 V2=H(K)
      J=VERTX(V2)
      IF(J)46,45,46
C
C      V2 IS IN A COMPONENT AND V1 IS NOT IN ANY COMPONENT.
C     ADD V1 TO THE COMPONENT.
C
   46 EDGE(K)=J
      VERTX(V1)=J
      GO TO 49
C
C      V1 AND V2 ARE NOT IN ANY COMPONENT.
C     FORM A NEW COMPONENT.
C
   45 C=C+1
      EDGE(K)=C
      VERTX(V1)=C
      VERTX(V2)=C
   49 M=M+1
   50 IF(M-N+1)51,52,51
   51 IF(KZ-E)10,52,10
   52 RETURN
      END
```

```
£IBFTC SORT
      SUBROUTINE SORT(N,K,NO)
C
C     SUBROUTINE SORT USES HEAP SORT FOR SORTING THE EDGE SET
C     OF THE GRAPH IN NONDECREASING ORDER OF WEIGHTS.
C     IN EVERY CALL TO THE SUBROUTINE THE SUBSCRIPT OF THE
C     SMALLEST EDGE NOT YET PICKED IS RETURNED.
C     K(I) DENOTES THE WEIGHT OF THE I&TH EDGE.
C     N DENOTES THE NUMBER OF EDGES.
C     NUM DENOTES THE SUBSCRIPT OF THE SMALLEST EDGE NOT
C     YET PICKED
C
      INTEGER R
      DIMENSION K(3000),NO(3000)
    1 L=N/2+1
      R=N
    2 IF(L-1)11,11,12
   12 L=L-1
      KK=K(L)
      KN=NO(L)
    3 J=L
    4 I=J
      J=2*J
      IF(J-R)5,6,8
    5 IF(K(J)-K(J+1))13,6,6
   13 J=J+1
    6 IF(KK-K(J))7,7,8
    7 K(I)=K(J)
      NO(I)=NO(J)
      GO TO 4
    8 K(I)=KK
       NO(I)=KN
      GO TO 2
   11 KK=K(R)
      KN=NO(R)
      K(R)=K(1)
      NO(R)=NO(1)
      R=R-1
      IF(R-1)14,14,3
   14 K(1)=KK
      NO(1)=KN
      RETURN
      END
```

APPENDIXII. MCILROY&S IMPLEMENTATION OF MST ALGORITHM

```
£IBFTC SPTRE
      SUBROUTINE SPTRE(F,H,N,E,EDGE,C,W)
C
C     MCILROY&S IMPLEMENTATION OF KRUSKAL ALGORITHM.
C     MCILROY&S IMPLEMENTATION GROWS A SPANNING TREE T FOR A GIVEN
C     UNDIRECTED GRAPH  OF N VERTICES AND E EDGES.
C     IF THE INPUT GRAPH IS DISCONNECTED A SPANNING FOREST
C     WILL BE GENERATED.
C     THE TWO TERMINAL VERTICES OF THE I&TH EDGE ARE STORED
C     IN ARRAYS F(I) AND H(I) AND ITS WEIGHT IS STORED IN W(I).
C     PRED(I) CONTAINS THE PREDECESSOR OF THE VERTEX VI IN THE
C     COMPONENT TREE.
C     NUM(I) DENOTES THE NUMBER OF VERTICES IN THE COMPONENT
C     TREE ROOTED AT VERTEX VI.
C
      INTEGER C,E,EDGE(3000),F(3000),H(3000),W(3000)
      INTEGER PRED(100),NUM(100),VI,VJ
      COMMON/MAC/ PRED,NUM
      COMMON/RAMS/IFLG
      DO 1 L=1,N
      PRED(L)=0
    1 NUM(L)=1
      DO 2 L=1,E
      NO(L)=L
    2 EDGE(L)=0
      IFLG=0
      NC=N
      NE=0
    3 NE=NE+1
C
C     PICK THE NEXT EDGE.
C
      CALL SORT(W,E,K)
      VI=F(K)
C
C     FIND THE LABEL OF THE COMPONENT CONTAINING VERTEX VI.
C
      CALL FIND(VI,LABELI)
      VJ=H(K)
C
C     FIND THE LABEL OF THE COMPONENT CONTAINING VERTEX VJ.
C
      CALL FIND(VJ,LABELJ)
C
C     IF THEY BELONG TO THE SAME COMPONENT REJECT THE EDGE.
C     OTHERWISE MERGE THE TWO COMPONENTS.
C
      IF(LABELI.EQ.LABELJ) GO TO 4
      CALL MERGE(LABELI,LABELJ)
```

```
C
C      REDUCE THE NUMBER OF COMPONENTS  BY 1.
C
       NC=NC-1
       EDGE(K)=0
C
C      IF THE GRAPH IS CONNECTED OR IF THE EDGE SET IS
C      EXHAUSTED STOP, ELSE PICK THE NEXT EDGE.
C
     4 IF(NE.LT.E.AND.NC.GT.1) GO TO [illegible]
       C=NC
       RETURN
       END

£IBFTC MERGE
       SUBROUTINE MERGE(I,J)
C
C       SUBROUTINE MERGE GRAFTS THE SMALLER OF THE TWO
C      SUBTREES TO THE LARGER SUBTREE.
C
       INTEGER PRED(100),NUM(100)
       COMMON/MAC/ PRED,NUM
       IDMY=NUM(I)+NUM(J)
       IF(NUM(I).GT.NUM(J)) GO TO 2
       NUM(J)=IDMY
       PRED(I)=J
       RETURN
     2 NUM(I)=IDMY
       PRED(J)=I
       RETURN
       END
```

```
$IBFTC FIND
      SUBROUTINE FIND(I,LABEL)
C
C      SUBROUTINE FIND RETURNS THE LABEL ( ROOT VERTEX )
C     OF THE TREE CONTAINING VERTEX VI.
C
      INTEGER STACK(100),STACKP,TOP,PRED(100),NUM(100)
      COMMON/MAC/ PRED,NUM
      IPTR=I
      STACKP=0
C
C     FIND THE ROOT AND STACK THE PATH FROM VI TO ROOT.
C
    1 IF(PRED(IPTR).EQ.0) GO TO 2
      STACKP=STACKP+1
      STACK(STACKP)=IPTR
      IPTR=PRED(IPTR)
      GO TO 1
C
C     MAKE ALL THE VERTICES IN THE PATH FROM I TO ROOT , IMMEDIATE
C     SUCCESSORS OF THE ROOT.
C
    2 IF(STACKP.LE.1) GO TO 3
      STACKP=STACKP-1
      TOP=STACK(STACKP)
      PRED(TOP)=IPTR
      GO TO 2
    3 LABEL=IPTR
      RETURN
      END
```

```
$IBFTC SORT
      SUBROUTINE SORT(K,N,NUM)
C
C     SUBROUTINE SORT USES HEAP SORT FOR SORTING THE EDGE SET
C     OF THE GRAPH IN NONDECREASING ORDER OF WEIGHTS.
C     IN EVERY CALL TO THE  SUBROUTINE THE SUBSCRIPT OF THE
C     SMALLEST EDGE NOT YET PICKED IS RETURNED.
C     K(I) DENOTES THE WEIGHT OF THE I-TH EDGE.
C     N DENOTES THE NUMBER OF EDGES.
C     NUM DENOTES THE SUBSCRIPT OF THE SMALLEST EDGE NOT
C     YET PICKED.
C
      INTEGER R
      DIMENSION NO(2000),K(2000)
      COMMON /RAMS/ IFLG
      IF(IFLG)20,20,4
   20 CONTINUE
      DO 21 I=1,N
   21 NO(I)=I
```

```
 1  L=N/2+1
    R=N
 2  IF(L-1)11,11,12
12  L=L-1
    KK=K(L)
    NN=NO(L)
 3  J=L
    IF(IFLG)4,4,15
15  NUM=NO(1)
    RETURN
 4  I=J
    J=2*J
    IF(J-R)5,6,8
 5  IF(K(J)-K(J+1))6,13,13
13  J=J+1
 6  IF(KK-K(J))8,7,7
 7  K(I)=K(J)
    NO(I)=NO(J)
    GO TO 4
 8  K(I)=KK
    NO(I)=NN
    GO TO 2
11  KK=K(R)
    NN=NO(R)
    IF(IFLG)16,16,17
16  IFLG=1
17  K(R)=K(1)
    NO(R)=NO(1)
    R=R-1
    IF(R-1)14,14,3
14  K(1)=KK
    NO(1)=NN
     RETURN
    END
```

APPENDIX III. THE PRIM AND DIJKSTRA ALGORITHM WITH BINARY TREE SORT.

```
C
C
      SUBROUTINE DYTRE(D,N,S,PRED)
£IBFTC DYTRE
C
C     THE PRIM AND DIJKSTRA ALGORITHM STARTS WITH THE GIVEN
C     VERTEX S AS THE STARTING VERTEX.
C     THE ALGORITHM DEVELOPS AN MST BY ADDING THE
C     VERTEX WITH THE SMALLEST LABEL TO THE SUBTREE IN
C     EACH ITERATION.
C     THE INPUT GRAPH WITH N VERTICES IS STORED IN THE
C     WEIGHT MATRIX D.
C     LABL(I) DENOTES THE LABEL ATTACHED TO THE VERTEX I.
C     VECT(I)=1 IF VERTEX I IS PERMANENTLY LABELED.
C     PRED(J) DENOTES THE PREDECESSOR OF VERTEX J IN THE
C     SUBTREE.
C
      INTEGER D(100,100),P,S,VECT(100),Z,PRED(100),HEAP(256),DIR(256)
      DIMENSION LABL(100)
      COMMON HEAP,DIR,INDX
      DO 6 L=1,N
      LABL(L)=9999
      PRED(L)=0
    6 VECT(L)=0
      CALL INTG(N)
      LABL(S)=0
      PRED(S)=0
      VECT(S)=1
      I=S
   10 M=9999
      P=0
      DO 18 J=1,N
      IF(VECT(J)-1)19,18,19
   19 Z=LABL(I)+D(I,J)
      IF(Z-LABL(J))20,18,18
   20 LABL(J)=Z
      PRED(J)=I
      P=J
      CALL ADD(J,Z)
   18 CONTINUE
      IF(P)24,24,21
   21 CALL PICK(P,4)
      CALL DELET(P)
      I=P
      VECT(P)=1
      GO TO 10
   24 CONTINUE
      RETURN
      END
```

```
£IBFTC INTG
      SUBROUTINE INTG(N)
C
C     SUBROUTINE INTG INITIALIZES THE BINARY SORT TREE
C     WITH LABEL OF EACH VERTEX = 9999.
C
      INTEGER HEAP(256),DIR(256)
      COMMON HEAP,DIR,INDX
      N2=1
   10 N2=N2*2
      IF(N2-N)10,20,20
   20 INDX=N2+N2
      DO 30 I=N2,INDX
      HEAP(I)=9999
   30 DIR(I)=0
      INDX=N2-1
      DO 40 I=1,INDX
      HEAP(I)=9999
   40 DIR(I)=-1
      RETURN
      END
£IBFTC ADD
      SUBROUTINE ADD(I,LABLT)
C
C     SUBROUTINE ADD REPLACES THE LABEL OF THE VERTEX I
C     IN THE BINARY SORT TREE BY LABLT.
C
      INTEGER HEAP(256),DIR(256)
      COMMON HEAP,DIR,INDX
      IPTR=I+INDX
      HEAP(IPTR)=LABLT
   10 JPTR=IPTR/2
      KPTR=IPTR-JPTR*2
      IF(HEAP(JPTR)-HEAP(IPTR))30,30,20
   20 HEAP(JPTR)=HEAP(IPTR)
      DIR(JPTR)=KPTR*2-1
      IPTR=JPTR
      IF(JPTR-1)30,30,10
   30 RETURN
      END
```

```
£IBFTC PICK
      SUBROUTINE PICK(I,LABLT)
C
C     SUBROUTINE PICK  RETURNS THEVERTEX I WITH THE
C     SMALLEST LABEL AND ITS LABEL, LABLT.
C
      INTEGER HEAP(256),DIR(256)
      COMMON HEAP,DIR,INDX
      IPTR=1
   10 J=DIR(IPTR)
      IF(J)20,40,30
   20 IPTR=IPTR*2
      GO TO 10
   30 IPTR=IPTR*2+1
      GO TO 10
   40 I=IPTR-INDX
      LABLT=HEAP(I)
      RETURN
      END
£IBFTC DELET
      SUBROUTINE DELET(I)
C
C     SUBROUTINE DELET DELETES THE VERTEX I FROM THE
C     BINARY SORT TREE BY MODIFYING ITS LABEL TO 9999.
C
      INTEGER HEAP(256),DIR(256)
      COMMON HEAP,DIR,INDX
      JPTR=INDX+I
      IPTR=JPTR
      HEAP(IPTR)=9999
   10 IPTR=JPTR
      JPTR=IPTR/2
      KPTR=IPTR-JPTR*2
      LEFT=DIR(JPTR)
      IF(KPTR)21,21,22
   21 IF(LEFT)20,20,40
   22 IF(LEFT)40,40,20
   20 IDMY=IPTR-LEFT
      IF(HEAP(IDMY)-HEAP(IPTR))30,35,35
   35 DIR(JPTR)=KPTR+KPTR-1
      HEAP(JPTR)=HEAP(IPTR)
      IF(JPTR-1) 40,40,10
   30 DIR(JPTR)=1-KPTR*2
      HEAP(JPTR)=HEAP(IDMY)
      IF(JPTR-1)40,40,10
   40 RETURN
      END
```

APPENDIX IV. MSA ALGORITHM

```
£IBFTC MSA
      SUBROUTINE MSA(MAT,MTREE,N)
C
C     PROGRAMM FOR FINDING MSA FOR A WEIGHTED DIGRAPH.
C     MAT REPRESENTS THE DIGRAPH GO IN THE WEIGHT MATRIX FORM,
C     MATI CONTAINS THE INITIAL VERTEX OF  EACH EDGE IN GI
C     CORRESPONDING TO DIGRAPH GO, AND MATJ THE TERMINAL VERTEX.
C     INOD GIVES A MAPPING BETWEEN THE ROWS AND COLUMNS OF
C     THE MATRIX AND THE VERTICES OF GI.
C     PRED CONTAINS THE  PREDECESSOR LIST REPRESENTATION OF
C     THE MIS AND, MTREE THE MERGER TREE.
C     AVL CONTAINS THE SUCCESSOR LIST OF THE  INTERNAL
C     VERTICES OF THE MERGER TREE.
C     N IS THE NUMBER OF VERTICES IN DIGRAPH GO,NMAX THE NUMBER OF
C     VERTICES IN THE MTREE AT ANY GIVEN TIME, AND
C     NOV THE NUMBER OF VERTICES IN GI.
C
      INTEGER STAK(70),PRED(70,2),AVL(500,2),STAKP,VERTX,TOP,PAVL
      DIMENSION MAT(70,70),MATI(70,70),MATJ(70,70),MTREE(150,5),INOD(70)
      COMMON /NMAXM/ NMAX ,IROOT
      PAVL=1
      NAVL=500
      DO 400 I=1,NAVL
      AVL(I,1)=0
  400 AVL(I,2)=I+1
      AVL(NAVL,2)=-1
  200 CONTINUE
      NMAX=0
      NOV=0
      NODE=N
C
C     SET THE INITIAL AND TERMINAL VERTEX MATRICES AND
C     INITIALIZE INOD AND LABL.
C
      DO 210 I=1,N
      INOD(I)=I
      PRED(I,2)=0
      DO 210 J=1,N
      MATI(I,J)=I
  210 MATJ(J,I)=I
C
C     FIND THE MINIMUM ENTRY IN EACH COLUMN AND SUBTRACT
C     IT FROM THE COLUMN.
C
      DO 260 I=1,N
      MIN=9999
      IF(INOD(I))250,220,220
  220 DO 240 J=1,N
      IF(MAT(J,I)-MIN)230,240,240
```

```
  230 MIN=MAT(J,I)
      IMIN=J
      IF(MIN) 240,251,240
  240 CONTINUE
      DO 250 J=1,N
  250 MAT(J,I)=MAT(J,I)-MIN
C
C     FORM THE PREDECESSOR LIST FOR MIS G0&.
C     INITIALIZE THE MERGER TREE MTREE.
C
  251 MTREE(I,1)=-1
      PRED(I,1)=IMIN
      NMAX=NMAX+1
      NOV=NOV+1
      MTREE(I,2)=INOD(IMIN)
      MTREE(I,3)=MATI(IMIN,I)
      MTREE(I,4)=MATJ(IMIN,I)
  260 MTREE(I,5)=MIN
C
C     PHASE I
C     CONSTRUCT MERGER TREE
C
  500 VERTX=1
C
C     LABL(TOP)=0 INDICATES THAT THE VERTEX IS ENCOUNTERED
C     FOR THE FIRST TIME.
C     LABL(TOP) .GT. ZERO INDICATES THAT THE VERTEX HAS APPEARED
C     FOR THE SECOND TIME IN THE PATH. A CYCLE IS IDENTIFIED.
C
   10 IF(PRED(VERTX,2))20,30,20
   20 VERTX=VERTX+1
      IF(VERTX-N)10,10,450
  450 DO 460 II=1,N
      IF(INOD(II))460,460,480
  480 PRED(II,2)=0
  460 CONTINUE
      IF(NOV-1)490,490,500
   30 STAKP=1
      STAK(STAKP)=PRED(VERTX,1)
      PRED(VERTX,2)=1
   40 TOP=STAK(STAKP)
      IF(PRED(TOP,2))70,50,60
   50 STAKP=STAKP+1
      STAK(STAKP)=PRED(TOP,1)
      PRED(TOP,2)=STAKP
      GO TO 40
   70 PRED(VERTX,2)=-1
   71 PRED(TOP,2)=-1
      STAKP=STAKP-1
      IF(STAKP)20,20,80
   80 TOP=STAK(STAKP)
      GO TO 71
C
```

```
C      MERGE THE ROWS
C
   60 NM=PRED(TOP,2)
      NUM=STAK2
      IROW=STAK(NUM)
      DO 130 I = 1,NODE
      MIN=9999
      IMIN=STAK(1)
      DO 120 J = NM,NUM
      IDMY=STAK(J)
      IF(MAT(I,IDMY)-MIN)110,110,120
  110 IMIN = IDMY
      MIN=MAT(I,IDMY)
  120 CONTINUE
      MAT(I,IROW)=MIN
      MATI(I,IROW)=MATI(I,IMIN)
  130 MATJ(I,IROW)=MATJ(I,IMIN)
C
C      MERGE THE COLUMNS
C
      DO 160 J=1,NODE
      MIN=9999
      IMIN=STAK(NUM)
      DO 150 I=NM,NUM
      IDMY=STAK(I)
      IF(MAT(IDMY,J)-MIN)140,140,150
C
C      MODIFY THE PREDECESSOR LIST
C
  140 IMIN=IDMY
      MIN=MAT(IDMY,J)
      IF(MIN)150,142,150
  142 IF(PRED(J,1)-IDMY)150,143,150
  143 PRED(J,1)=IR0W
  144 MIN=MAT(IDMY,J)
  150 CONTINUE
      MAT(IROW,J)=MIN
      MATJ(IROW,J)=MATJ(IMIN,J)
  160 MATI(IROW,J)=MATI(IMIN,J)
      MAT(IROW,IROW)=9999
      NMAX=NMAX+1
      NOV=NOV-NUM+NM
C
C      FORM THE MERGER TREE
C
      MTREE(NMAX,1)=PAVL
      DO 170 I=NM,NUM
      IPAVL=PAVL
      IDMY=STAK(I)
      ID=IDMY
      IDMY=INOD(IDMY)
      AVL(PAVL,1)=IDMY
      MTREE(IDMY,2)=NMAX
```

```
      INOD(ID)=-INOD(ID)
  170 PAVL=AVL(PAVL,2)
      AVL(IPAVL,2)=0
      NM=NM+1
      INOD(IROW)=NMAX
C
C     FIND THE MINIMUM ENTRY IN THE MERGED COLUMN
C
  270 MIN=9999
      DO 310 I = 1,N
      IF(INOD(I))310,310,290
  290 IF(MAT(I,IROW)-MIN)300,300,310
  300 MIN=MAT(I,IROW)
      IMIN=I
  310 CONTINUE
C
C     SUBTRACT THE MINIMUM ENTRY FROM THE COLUMN
C
      DO 320 I=1,N
      IF(INOD(I))320,320,330
  330 MAT(I,IROW)=MAT(I,IROW)-MIN
  320 CONTINUE
C
C     ASSIGN THE WEIGHT AND TAG TO THE EDGE INCIDENT INTO
C     NMAX IN THE MERGER TREE.
C     MERGER TREE.
C
      MTREE(NMAX,2)=INOD(IMIN)
      MTREE(NMAX,3)=MATI(IMIN,IROW)
      MTREE(NMAX,4)=MATJ(IMIN,IROW)
      MTREE(NMAX,5)=MIN
C
C     ASSIGN PREDECESSOR TO NMAX IN G&I+1
C
      PRED(IROW,1)=IMIN
      PRED(IROW,2)=-1
      GO TO 70
C
C     PHASE II
C     LABEL MERGER TREE
C
  490 CONTINUE
C
C     FIND THE OPTIMAL ROOT VERTEX.
C
      MTREE(NMAX,5)=0
      STAKP=1
      II=NMAX
      IPTR=MTREE(II,1)
C
C     FIND  PATH LENGTH FROM II TO ALL TERMINAL VERTICES.
C     LABEL VERTICES OF MTREE.
C
```

```
  600 IDMY=AVL(IPTR,1)
      STAK(STAKP)=IDMY
      STAKP=STAKP+1
      MTREE(IDMY,5)=MTREE(IDMY,5)+MTREE(II,5)
      IF(AVL(IPTR,2))620,620,610
  610 IPTR=AVL(IPTR,2)
      GO TO 600
  620 STAKP=STAKP-1
      IF(STAKP)650,650,630
  630 IPTR=STAK(STAKP)
      IF(MTREE(IPTR,1))620,620,640
  640 II=IPTR
      IPTR=MTREE(IPTR,1)
      GO TO 600
  650 CONTINUE
C
C     PICK THE VERTEX WITH THE LARGEST LABEL IN THE MTREE.
C
      STAKP=1
      STAK(STAKP)=NMAX
      STAKP=STAKP+1
      MAX=0
      IMAX=1
      DO 670 I=1,N
      IF(MTREE(I,5)-MAX)670,670,660
  660 MAX=MTREE(I,5)
      II=I
  670 CONTINUE
C
C     PHASE III
C     MERGER TREE MARKING
C
      IROOT=II
      II=NMAX
      STAKP=0
      STAKP=STAKP+1
      STAK(STAKP)=II
C
C     MARK THE EDGES IN THE PATH FROM II TO IROOT
C
  700 IF(MTREE(II,1))710,701,701
  701 MTREE(IROOT,3)=-1
      IROOT=MTREE(IROOT,2)
      IF(IROOT-II)700,710,710
C
C     VISIT THE NEXT SUCCESSOR
C
  710 IPTR=MTREE(II,1)
      IF(IPTR)730,730,720
  720 IDMY=AVL(IPTR,1)
      MTREE(II,1)=AVL(IPTR,2)
      II=IDMY
      STAKP=STAKP+1
```

```
      STAK(STAKP)=IDMY
      IF(MTREE(IDMY,5))710,72 ,721
  721 IROOT=MTREE(IDMY,4)
      GO TO 700
C
C     IF NO SUCCESSOR LEFT UNSTACK THE VERTEX
C
  730 STAKP=STAKP-1
      IF(STAKP)790,790,750
  750 II=STAK(STAKP)
      GO TO 710
  790 RETURN
      END
```

A 30007

EE- 1974 - M- PAT-MIN